
CROSSTALK
December 2005 · The Journal of Defense Software Engineering Vol. 18 No. 12
TOTAL CREATION of a SOFTWARE PROJECT

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2005		**00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk: The Journal of Defense Software Engineering. Volume 18, Number 12, December 2005	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

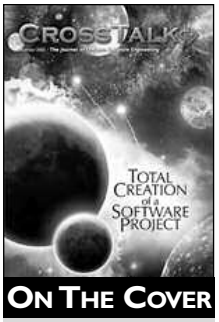
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**32**	

Policies, News, and Updates

Total Creation of a Software Project

Best Practices

Open Forum



ON THE COVER
Cover Design by Kent Bingham.

Additional art services provided by Janna Jensen. jensendesigns@aol.com

Departments

The CrossTalk staff would like to wish you and yours the very best this holiday season and the happiest of New Years.

CROSSTALK

76 SMXG Co-Sponsor	*Kevin Stamey*
309 SMXG Co-Sponsor	*Randy Hill*
402 SMXG Co-Sponsor	*Bob Zwitch*
DHS Co-Sponsor	*Joe Jarzombek*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF) and the U.S. Department of Homeland Security (DHS). USAF co-sponsors are the Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor is the National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send subscription correspondence and changes of address to the following address or e-mail us, or use the form on p. 27.

309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Endorsements and Trademarks: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk *Online Services:* See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857, or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Successful Software Is an Epic Production

From conception to completion, good software requires a more-than-just-code mindset. Good software is the product of the successful execution of hundreds of sound project and process practices. The list, which requires strict adherence to, might seem staggering to the casual observer: documented policies and procedures; requirements management; detailed project planning; system design; configuration management; peer reviews; tracking and oversight; collecting and managing with metrics; accurate useable documentation; and thorough testing against requirements at the code, subsystem, and system levels, just to name a few.

At the next level, good software becomes excellent software through institutionalized quantitative processes that are measured, analyzed, and optimized over time. At Hill Air Force Base, Utah, the 309th Software Maintenance Group (309 SMXG) has achieved this level by embracing continuous process improvement. The 309 SMXG has an established Capability Maturity Model® Integration process capability and is adding new capabilities as it strives to implement AS9100 quality management standard requirements.

Yet it doesn't stop there. As important as process is, you can't produce software without people. Successful execution occurs at the hands of trained and cross-trained, experienced, mentored, and motivated people. As systems become larger and more complicated, employees with greater years of experience become even more important. Like Jim Collins said in his book "Good to Great," "...people aren't your most important asset; *the right people* are your most important asset."

Randy B. Hill

Ogden Air Logistics Center, Co-Sponsor

Software Development Is More Than Coding

This month we announce that nominations begin for the 2005 Top 5 U.S. Department of Defense Programs Awards. The Office of the Undersecretary of Defense is once again recognizing those programs most successful at applying systems engineering best practices in the management, development, and integration of hardware and software. This award is in line with CROSSTALK'S theme this month: total creation of a software project. Leaders in the software community are constantly reminding us that good software development involves so much more than developing code. All the players, including customers, must employ sound practices such as effective requirements definition, removing defects at their origin, measurement throughout, effective training, and communication.

We begin our theme section with *Correctness by Construction: A Manifesto for High Integrity Systems* by Martin Croxford and Dr. Roderick Chapman, who discuss how these practices apply to the entire development life cycle. Next, Granville Miller discusses a Microsoft approach to agile software development in *Agile Software Development for the Entire Project*. In *Eliminating Embedded Software Defects Prior to Integration Test*, Ted L. Bennett and Paul W. Wennberg discuss how testing software can be performed during the entire development cycle, even design.

In our supporting articles, Watts S. Humphrey's *Acquiring Quality Software* discusses using measurements to ensure quality, and in *Role of Human Emotions in Requirements Management*, Sreevalli Radhika. T. discusses how a requirements document might affect the entire development process. We wrap up our 18th volume with the Article Index on page 28, highlighting articles published in 2005.

Elizabeth Starrett

Associate Publisher

OFFICE OF THE UNDER SECRETARY OF DEFENSE

3000 DEFENSE PENTAGON
WASHINGTON, DC 20301-3000

ACQUISITION,
TECHNOLOGY
AND LOGISTICS

SEP 09 2005

MEMORANDUM FOR ALL DOD GOVERNMENT PROGRAM OFFICES

SUBJECT: 2005 TOP 5 U.S. DEPARTMENT OF DEFENSE PROGRAMS AWARDS

The Department of Defense's Executive Agent for Systems Engineering and the Systems Engineering Division of the National Defense Industrial Association are pleased to announce the search for the Top 5 U.S. Department of Defense Programs.

Many organizations are employing processes and practices that result in the successful delivery of programs to the Department of Defense. Looking at past winners of this award, it is apparent that successful programs have used well-defined and proven practices to develop, manage, and integrate hardware and software into deliverable systems. This award was restructured to recognize successful application of systems engineering best practices to programs.

The significant change to the award criteria from past years is the focus on systems engineering and integration for program success. Nominees will be evaluated as to their effective application of systems engineering fundamentals (sound life-cycle technical planning, use of event-based reviews to manage the technical baseline and control risk, proper application of technical authority, and independent subject-matter experts in the conduct of technical reviews) across the acquirer and supplier (and sub-tier supplier) domains.

Top 5 U.S. Department of Defense Programs, with the DoD and industry-winning organizations, will be announced in the October 2006 issue of the National Defense Industrial Association's National DEFENSE magazine, and winners will receive their awards at the October 2006 NDIA Systems Engineering conference in San Diego, Calif.

To access the full announcement and the nomination forms, go to www.ndia.org, then to the Systems Engineering page under Divisions. Nominations must be received no later than January 30, 2006.

Robert Skalamera
Deputy Director, Systems Engineering
Enterprise Development
OUSD(AT&L) DS/SE/ED

Correctness by Construction: A Manifesto for High-Integrity Software

Martin Croxford and Dr. Roderick Chapman
Praxis High Integrity Systems

High-integrity software systems are often so large that conventional development processes cannot get anywhere near achieving tolerable defect rates. This article presents an approach that has delivered software with very low defect rates cost-effectively. We describe the technical details of the approach and the results achieved, and discuss how to overcome barriers to adopting such best practice approaches. We conclude by observing that where such approaches are compatible and can be deployed in combination, we have the opportunity to realize the extremely low defect rates needed for high integrity software composed of many million lines of code.

The National Institute of Standards and Technology (NIST) reported in 2002 that low quality software costs the U.S. economy $60 billion per year [1]. According to the aptly named "Chaos Report," only one quarter of software projects are judged a success [2]. Software defects are accepted as inevitable by both the software industry and the long-suffering user community. In any other engineering discipline, this defect rate would be unacceptable. But when safety and security are at stake, the extent of current software vulnerability is unsustainable.

Recent research on this issue has been conducted on behalf of the National Cyber Security Partnership, formed in 2003 in response to the White House National Strategy to Secure Cyberspace [3]. The partnership's Secure Software Task Force report states the following:

> Software security vulnerabilities are often caused by defective specification, design, and implementation. Unfortunately today, common development practices can often leave numerous defects and resulting vulnerabilities in the complex artifact that is delivered software. To have a secure U.S. cyber infrastructure, the supporting software must contain few, if any, vulnerabilities. [4]

The report goes on to recommend adoption of software development processes that can measurably reduce software specification, design, and implementation defects. It identifies three software engineering practices as examples that satisfy this recommendation. This article describes one of these examples, *Correctness by Construction* (CbyC), which originates from Praxis High Integrity Systems.

Maturity of Approach

The CbyC approach has two primary goals: to deliver software with defect rates an order of magnitude lower than current best commercial practices in a cost-effective manner, and to deliver durable software that is resilient to change throughout its life cycle.

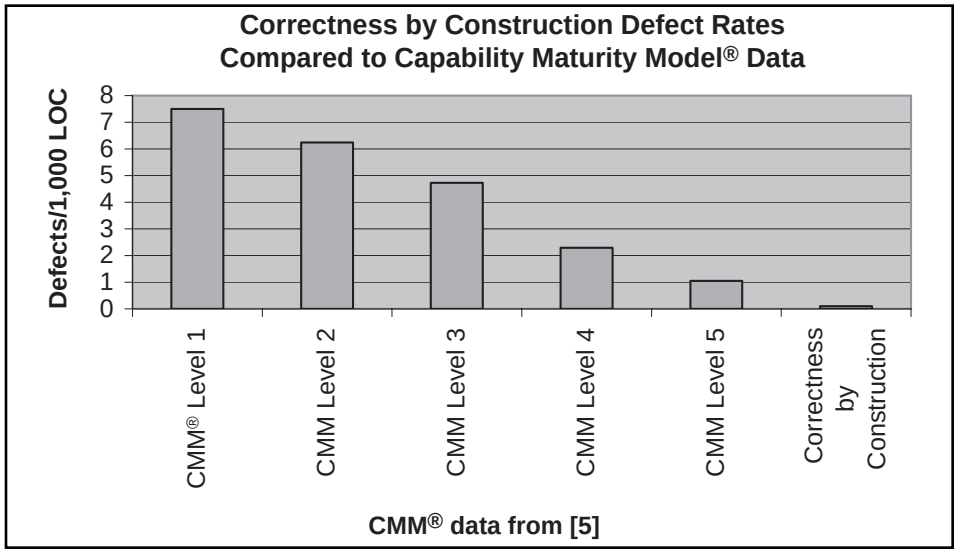
Elements of the CbyC approach have been used for more than 15 years to produce software with very low defects mainly for safety-critical applications, but more recently for security-critical applications. The approach has evolved over time and now applies to the entire systems development life cycle, from validation of the concepts of operation to preserving correctness properties during long-term maintenance.

CbyC has delivered software with defect rates of less than 0.1 defects/1,000 source lines of code (SLOC) with good productivity: up to around 30 LOC per day. The achieved defect rates compare very favorably with defect rates reported by Capability Maturity Model® Level 5 organizations of 1 defect/1,000 LOC [5]. The comparative rates are shown in Figure 1. It is, of course, true that other approaches have also succeeded in delivering similarly low defect rates, however, it is rare to also deliver good productivity (since low defect rates are often the result of extensive, expensive debugging and testing).

As well as realizing low defect rates, the CbyC approach has also proved to be highly cost-effective during both development and maintenance. Metrics for five fully deployed projects are shown in Figure 2 (see page 7).

Given that CbyC and other best-practice approaches cited in the National Cyber Security Summit Task Force report [4] have been used so successfully for a number of years, you may ask: Why are these approaches not in more widespread use, especially where high levels of assurance are required?

Figure 1: *Correctness by Construction Defect Rates Comparison*



® Capability Maturity Model is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Before considering the barriers to adoption of best practices, it is necessary to examine the nature of CbyC. A more detailed white paper on CbyC [6] is freely available from the authors.

Fundamental Principles

The primary goals of very low defect rate and very high resilience to change are realized in CbyC by two fundamental principles: to make it difficult to introduce errors in the first place, and to detect and remove any errors that do occur as early as possible after introduction.

The key to implementing these principles is to introduce sufficient precision at each step of the software development to enable reasoning about the correctness of that step – reasoning in the sense that an argument for correctness can be established either by review or using tool support. The aim is to demonstrate or argue the software correctness in terms of the manner in which it has been produced (*by construction*) rather than just by observing operational behavior.

It is the use of precision that differentiates approaches such as CbyC from others in common use. Typically, software development approaches endure a lack of precision that makes it very easy to introduce errors, and very hard to find those errors early. Evidence for this may be found in the common tendency for development life cycles to migrate to an often-repeating *code-test-debug* phase, which can lead to severe cost and timescale overruns.

Conversely, the rigor and precision of the CbyC approach means that the requirements are more likely to be correct, the system is more likely to be the correct system to meet the requirements, the implementation is more likely to be defect-free, and upgrades are more likely to retain the original correctness properties.

Achieving the Fundamental Principles

The principles of making it difficult to introduce defects and making it easy to detect and remove errors early are achieved by a combination of the following six strategies:

1. **Using a sound, formal notation for all deliverables.** For example, using Z [7] for writing specifications so it is impossible to be ambiguous, or using SPARK [8] to write the code so it is impossible to introduce errors such as buffer overflows.
2. **Using strong, tool-supported methods to validate each deliverable.** For example, carrying out proofs of formal specifications and static analysis of code. This is only possible where formal notations are used (strategy No. 1).
3. **Carrying out small steps and validating the deliverable from each step.** For example, developing a software specification as an elaboration of the user requirements, and checking that it is correct before writing code. For example, building the system in small increments and checking that each increment behaves correctly.
4. **Saying things only once.** For example, by producing a software specification that says what the software will do and a design that says how it will be structured. The design does not repeat any information in the specification, and the two can be produced in parallel.
5. **Designing software that is easy to validate.** For example, writing simple code that directly reflects the specification, and testing it using tests derived systematically from that specification.
6. **Doing the hard things first.** For example, by producing early prototypes to test out difficult design issues or key user interfaces.

These six principles are not in themselves difficult to apply, and may even appear obvious. However, in the authors' experience, many software development projects fail to deliver against many, if any, of these principles.

Requirements Engineering

At the requirements step (a source of half of project failures [2]), a clear distinction is made between user requirements, system specifications, and domain knowledge. CbyC uses *satisfaction arguments* to show that each user requirement can be satisfied by an appropriate combination of system specification and domain knowledge. The emphasis on domain knowledge is key – half of all requirements errors are related to domain [9] – yet the vast majority of requirements processes do not explicitly address issues in the domain.

Formal Specification and Design

Using mathematical (or formal) methods and notations to define the specification and high-level design provide a precise description of behavior and a precise model of its characteristics. This enables using tools to verify that the design meets its specification and that the specification meets its requirements.

Example languages used for formal specification in CbyC projects include Z and Communicating Sequential Processes [10].

Development

The CbyC approach applies rigor to all software development phases, including detailed design, implementation, and verification. As a result, static analysis tools can be used to produce evidence of correctness and completeness.

CbyC defines a software design methodology based on information flow that can be expressed using an unambiguous notation. This notation is contract-based, i.e., it is used to define both the abstract state and the information relationships across the software modules.

For the coding phase, the CbyC approach recommends using languages and tools that are most appropriate for the task at hand. Validation requirements play a large role in this choice: The selected languages must be amenable to verification and analysis so that the required evidence of correctness can be generated effectively. For high-integrity software modules, the SPARK programming language is especially suitable owing to its rigorous and unambiguous semantics.

Using mathematically verifiable programming languages such as SPARK opens the way for static analysis tools to provide proofs for absence of common runtime errors such as buffer overflows and using uninitialized variables. Being able to prove absence of runtime errors, rather than discovering a subset of them by testing, is critical to the achievement of very low defect rates.

Note that other programming languages have been used for CbyC projects, and that CbyC projects often have an element of mixed-language implementation. For example, C, C++, Structured Query Language (SQL), Ada '83 and Ada '95 have been used. However, such languages are intrinsically unsuitable for deep static analysis and are only ever used for the non-critical parts of the implementation.

Results

Experience from a wide variety of projects has confirmed that CbyC is both effective and economical due to the following:

1. Defects are removed early in the process when changes are cheap. Testing becomes a confirmation that the software works, rather than the point at which it must be debugged.
2. Evidence needed for safety or security certification is produced naturally as a byproduct of the process.
3. Early iterations produce software that

carries out useful functions and builds confidence in the project.

Figure 2 shows results from three safety-critical and two security-critical projects that have used elements of the CbyC approach. For all of these projects, the reported productivity figures are for the whole life cycle, from requirements to delivery.

The Ship/Helicopter Operating Limits Information System [11] was developed in 1997 and was the first project to be developed to the full degree of rigor required by the United Kingdom (UK) Ministry of Defence (MoD), Defence Standard 00-55 [12] at the highest safety integrity level.

The certification authority system to support the Multimedia Office Server (MULTOS) smart card operating system developed by Mondex International [13] was developed to the standards of the Information Technology Security Evaluation Criteria (ITSEC) Level E6[1], roughly equivalent to Common Criteria Evaluation Assurance Level (EAL) 7. The system had an operational defect rate of 0.04 defects/KLOC, yet was developed at a productivity of almost 30 LOC per day (three times typical industry figures).

CbyC was used in 2003 to develop a demonstrator biometrics system for the National Security Agency (NSA), aimed at showing that it is possible to produce cost-effective, high-quality, low-defect software conforming to the Common Criteria EAL 5 and above [14]. The software was subjected to rigorous independent reliability testing that identified zero defects and was developed at a productivity of almost 40 LOC/day.

These and other similar projects have demonstrated that the rigorous techniques employed by CbyC such as formal methods and proofs should no longer be viewed as belonging solely to academia, but can be used confidently and effectively in the commercial sector.

Project	Year	Size (SLOC)	Whole Life-Cycle Productivity (SLOC/day)	Defects (/1,000 SLOC)
CDIS[1]	1992	197,000	12.7	0.75
SHOLIS[2]	1997	27,000	7.0	0.22
MULTOS CA[3]	1999	100,000	28.0	0.04
A[4]	2001	39,000	11.0	0.05
NSA[5]	2003	10,000	38.0	0

Notes
[1] Real-time air traffic information system at the London Terminal Control Centre.
[2] Ship/Helicopter Operating Limits Information System developed to UK MoD Defence Standard 00-55 Safety Integrity Level 4 (highest).
[3] Certification authority for smart card operating system maintained by Mastercard.
[4] A UK military stores management system.
[5] NSA Tokeneer ID Station demonstrator biometrics system.

Figure 2: *Correctness By Construction Project Metrics*

Barriers to Adoption

Earlier, the question asked was why best practices such as CbyC and others referenced by [4] are not in widespread use. The authors contend that there are two kinds of barriers to the adoption of best practices.

First, there is often a cultural mindset or awareness barrier. Many individuals and organizations do not recognize or believe that it is possible to develop software that is low-defect, high-integrity, and cost-effective. This may simply be an awareness issue, in principle readily addressed by articles such as this. Or there may be a view that such best practices *could never work here* for a combination of reasons. These reasons are likely to include perceived capability of the staff, belief about applicability to the organization's product or process, prevalence of legacy software that is viewed as inherently inappropriate for such approaches, or concern about the disruption and cost of introducing new approaches.

Second, where the need for improvement is acknowledged and considered achievable, there are usually practical barriers to overcome such as how to acquire the necessary capability or expertise, and how to introduce the changes necessary to make the improvements.

Overcoming the Barriers

The barriers mentioned above are reasonable and commonplace, but not insurmountable. Overcoming them requires effort from suppliers, procurers, and regulators and involvement at the individual, project, and organizational level. Typically, strong motivation and leadership will be required at a senior management level where the costs to the business of poor quality (high defects, low productivity, and lack of resilience to change) are most likely to be experienced.

The authors have worked with a number of organizations to overcome these barriers. For example, the MULTOS system was delivered to Mondex International, along with three weeks training in the techniques used to develop it, and three weeks of part-time mentoring. Mondex has since successfully maintained the system – to the same development standards – with no further support from Praxis. The NSA system was successfully adapted by summer interns during a 12-week placement after minimal training in the techniques used to develop it.

The key to successful adoption of CbyC is the adoption of an engineering mindset. In particular, decisions on process, methods, and tools for software development need to be premised on the basis of logic and precision (for example, by asking, "How does this choice help me meet one of the six strategies of CbyC?"), rather than on fashion (characterized by questions such as, "How many developers already know this particular technology?").

Procurers have a role in overcoming barriers to best practices by demanding low defects. Regulation also has a role to play in requiring best practices; this is already happening within the security sector, for example Common Criteria EAL 5 and above, and within the safety sector, particularly in Europe, for example in the UK Civil Aviation Authority regulatory objectives for software [15] and the UK MoD safety standard 00-55 [12].

Maximizing the Benefit

Given the massive size of many software systems – some of which need to be high integrity – even a defect rate of 0.04 defect per KLOC may result in an unacceptably high number of faults. To address this, we need to employ a combination of compatible defect-prevention approaches.

One of the other identified approaches in the Secure Software Task Force report is the Team Software Process℠ (TSP℠) and Personal Software Process℠

℠ Team Software Process, Personal Software Process, TSP, and PSP are service marks of Carnegie Mellon University.

(PSP[SM]) from the Software Engineering Institute [16]. Since the focus of TSP/PSP is on improving the professional culture and working practices of individuals, teams, and management, and hence is largely independent of languages, tools, and methodologies that are used, the deployment of CbyC within an environment such as TSP/PSP is highly feasible and has already been demonstrated: A CbyC practitioner's results at a recent PSP training course were both defect-free and first to be completed. Given that the TSP/PSP approach has also demonstrated a very low defect rate, the combination of these approaches offers the best opportunity to realize the orders of magnitude reduction in a defect rate that are needed for a multi-million LOC high-integrity software subsystem.

Conclusions

Critical software subsystems are now large enough such that conventional development processes cannot get anywhere near reducing defect rates to tolerable levels.

A mature approach based on applying rigor and precision to each phase of the life cycle has demonstrated over the past 15 years that major improvements in defect rate are attainable while maintaining productivity levels and overall cost-effectiveness.

Where such compatible approaches can be deployed in combination, we can at last see extremely low defect rates needed for high-integrity software composed of many million lines of code.◆

Acknowledgements

The authors acknowledge contributions from Brian Dobbing, Peter Amey, and Anthony Hall of Praxis High Integrity Systems.

References

1. Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing. Ed. Dr. Gregory Tassey. RTI Project No. 7007.011. Washington, D.C.: National Institute of Standards and Technology, May 2002 <www.mel.nist.gov/msid/sima/sw_testing_rpt.pdf>.
2. Standish Group International. The Chaos Report. West Yarmouth, MA: Standish Group International, 2003 <www.standishgroup.com>.
3. National Cyber Security Partnership. "About the National Cyber Security Partnership." Washington, D.C.: NCSP, 18 Mar. 2004 <www.cyberpartnership.org/about-overview.html>.
4. National Cyber Security Task Force. "Improving Security Across the Software Development Life Cycle." Washington, D.C.: National Cyber Security Partnership, 1 Apr. 2004 <www.cyberpartnership.org/init-soft.html>.
5. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Reading, MA: Addison-Wesley, 2000.
6. Praxis High Integrity Systems. "Correctness by Construction: A White Paper." Issue 1.2, Jan. 2005. Please contact the authors for a copy of this paper.
7. Spivey, J.M. The Z Notation: A Reference Manual. 2nd ed. Prentice-Hall, 1992.
8. Barnes, J. High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley, 2003.
9. Hooks, Ivy F., and Kristin A. Farry. Customer Centered Products: Creating Successful Products Through Smart Requirements Management. 1st ed. New York: American Management Assoc., 11 Sept. 2000.
10. Hoare, C.A.R. Communicating Sequential Processes. Prentice-Hall, 1985.
11. King S., J. Hammond, R. Chapman, and A. Pryor. "Is Proof More Cost-Effective Than Testing?" IEEE Transactions on Software Engineering 26.8 (Aug. 2000) <www.praxis-his.com/pdfs/cost_effective_proof.pdf>.
12. United Kingdom Ministry of Defence. "Def. Stan. 00-55." Requirements for Safety Related Software in Defense Equipment Issue 2, Aug. 1997.
13. Hall, A., and R. Chapman R. "Correctness by Construction: Developing a Commercial Secure System." IEEE Software Jan./Feb. 2002 <www.praxis-his.com/pdfs/c_by_c_secure_system.pdf>.
14. National Security Agency. Fourth Annual High Confidence Software and Systems Conference Proceedings. Washington, D.C.: NSA, Apr. 2004.
15. United Kingdom Civil Aviation Authority. "CAP 670, Air Traffic Services Safety Requirements. Amendment 3." UKCAA, Sept. 1999.
16. Humphrey, W. Introduction to the Team Software Process. Addison-Wesley, 2000.

Note

1. Information about ITSEC and the Common Criteria can be found at <www.cesg.gov.uk/site/iacs/index.cfm>.

About the Authors

Martin Croxford is associate director for security with Praxis High Integrity Systems, a United Kingdom-based systems engineering company specializing in mission-critical systems. He is a chartered engineer with 15 years experience in the software industry. Croxford has worked on software development projects in a range of organizations, and as a software development manager has used Correctness by Construction to successfully deliver a multi-million dollar security-critical system.

Praxis High Integrity Systems
20 Manvers ST
BATH BA1 1PX
UK
Phone: (44) 1225-823794
Fax: (44) 1225-469006
E-mail: martin.croxford@praxis-his.com

Roderick Chapman, Ph.D., is product manager of SPARK with Praxis High Integrity Systems, specializing in the development of programming languages and static analysis tools for high integrity systems. He is a chartered engineer with more than a decade of experience in high integrity real-time systems. Chapman is internationally renowned for his work on verification of correctness properties of high integrity software. He has a Doctor of Philosophy in computer science.

Praxis High Integrity Systems
20 Manvers ST
BATH BA1 1PX
UK
Phone: (44) 1225-823763
Fax: (44) 1225-469006
E-mail: rod.chapman@praxis-his.com

Agile Software Development for the Entire Project

Granville Miller
Microsoft

Does an agile software development process require real organizational change or can an existing organization become more agile? How do the many traditional information technology (IT) roles such as the business analyst, architect, and tester become a more integrated part of an agile process? Some recent work [1] debunks the myths that agile processes require on-site customers, produce ad-hoc requirements and design, and cannot scale to large projects. This article furthers this work by introducing innovative techniques from a new agile process developed and used by projects within Microsoft. These techniques span the traditional IT roles such as the business analyst, project manager, architect, developer, tester, and release manager.

Many of today's more popular agile software development processes concentrate strictly on the developer and project manager. Traditional information technology (IT) roles such as business analysts, architects, and testers do not play a part in many of these agile processes. Yet, most software product and IT organizations have these roles or their equivalent. What is more, they are not ready to give up on them. On the contrary, these roles are becoming more valuable rather than less so as distributed development becomes more prevalent.

There are other practices such as the on-site customer, universal code ownership, pair programming, and stand up meetings that have proven barriers to widespread adoption of the more popular agile processes in many organizations. We have heard that it is mythical that these practices are required to be agile [1]. However, we have not been offered alternatives in a process form. This article introduces Microsoft Solutions Framework (MSF) for Agile Software Development, a context-based, agile software development process for building .NET applications [2]. This new process provides innovative techniques to extend agile software development to all of the traditional IT roles.

MSF for Agile Software Development is composed of a set of proven practices commonly used to build software at Microsoft. These practices have been collected in an agile form and used by teams both inside and outside of Microsoft. This process provides a set of practices that complement each other; that is, the sum of the practices is greater than each one used in isolation [3]. It also presents alternative practices to those commonly found in many agile processes.

The Agile Pattern

The core of any agile software development process is the way that it partitions and plans the work. Most agile processes share a similar method of planning or the *planning game* [4]. To start, a project is divided into time boxes or fixed periods in which *development* is done. These time boxes are called iterations. The iteration length is usually fixed between two to eight weeks, although really small projects have been known to set the iteration length in days or even hours.

> *"The personas describe usage patterns, knowledge, goals, motives, and concerns of a group of users. The key to good personas is that they are memorable and represent a set of typical customers."*

In each time box, we schedule work from two lists, our version of the product backlog [5]. The first is the scenario list that contains the names of scenarios (or scenario entries) that serve as placeholders for necessary functionality. The second is the quality-of-service requirement list that contains a list of requirements in areas such as performance, platform, or security. The scenarios and quality-of-service requirements in these lists are prioritized, and rough order-of-magnitude estimates are initially provided by the developers.

Scenarios and quality-of-service requirements are selected for the upcoming iteration and placed in the iteration plan (the equivalent of the sprint backlog [5]). The amount of work that is chosen is based upon the previous iteration's velocity. Once selected for iteration, more detailed scenario information is written by the business analyst. After the detailed information is provided, developers divide the scenarios into tasks and provide more detailed costs for the tasks. The costs are checked to make sure no developer is overloaded.

All of this planning occurs in a staggered manner. For example, our business analyst and project manager are working on planning iteration 1 in iteration 0. Developers spend a negligible portion of their time dividing the scenarios (and quality-of-service requirements) into tasks and choosing their tasks for the next iteration. However, most of their time is spent completing their tasks for the current iteration.

Development tasks are just one form of work breakdown that occurs. Testers and architects also create tasks as part of the iteration plan. These roles are integrally involved in ensuring that the solution is well architected and tested. They work in conjunction with the developers, business analysts, and project managers to ensure the system holds together. We will explore the nature of the architect and test roles later in this article.

Customer Collaboration Over Contract Negotiation[1]

There is no denying that the on-site customer, a customer that can work directly with the team to explain what is required of the system, is probably the best way to ensure project success. Unfortunately, most users have jobs other than guiding the delivery of a new system. It is rather ironic that the very thing that leads to a successful project is such a rare occurrence.

At Microsoft, lack of time is only one reason our users may not be able to be on-site. They may be located in a different city or even a different country. They may not be a part of our company at all in the case of commercial products. In any of these circumstances, our ability to interact with these users may be limited. When we obtain the opportunity to interact, we need to make the most of it. We also need to be able to communicate the essence of these interactions to the rest of the team.

Of course, this is exactly what the business analyst[2] is supposed to do in most organizations. In cases where travel is necessary to interact with users, they go. After all, nothing interesting happens in the office. We send these folks to meet with our customers because sending developers on frequent trips has an adverse affect on the project's velocity. However, customer knowledge should not be locked in a few people's heads. Instead, it should be shared with the entire team.

Sharing details of a customer visit is commonly performed in most organizations with trip reports. However, trip reports are an inadequate vehicle for providing anything more than a cursory insight into the customers. Instead, Microsoft utilizes a technique called *personas* as a basis for bringing the spirit of the customer to the entire development team [7]. Personas are respectful, fictitious people that represent groups of users. The personas describe usage patterns, knowledge, goals, motives, and concerns of a group of users. The key to good personas is that they are memorable and represent a set of typical customers.

Personas can also be compared to actors in use-case models [8]. An actor is an entity that interacts with the system. Human actors are instances of a role. The actor often contains very little information other than this role name. Therefore, while an on-site customer can usually provide us with better insight, an actor provides fewer details about the user community than a persona. In fact, actors make the assumption that all of the people that play a given role interact with the system in the same way.

Personas allow all of the members of the development team to obtain a deeper understanding of the user community. Design, development, and test decisions can often be made purely on the basis of a good persona. This allows the team to maintain velocity even when the business analyst is *on the road*. Personas must be constantly refined as new information is learned through interactions with the users. Posters of the personas can be found on the walls in the halls of the Microsoft campus.

In addition to writing the personas, the business analyst also generates the scenario entries in the scenario list. Once a scenario is scheduled for iteration, the business analyst writes up the details of that scenario. Personas are used in these scenario descriptions to show how a user would interact with the system. This provides the development team with an even deeper insight into the user community through understanding how the personas interact with the system.

Finally, there is no substitute for reviews of system functionality after key iterations with the customer. There are many vehicles for these reviews from actual working systems to storyboards with screen captures in cases where it is impossible to simulate the deployment environment in the area where the review is held. Experience at Microsoft has shown that using personas in conjunction with scenarios leads to fewer changes resulting from these reviews than when personas were not used. Ultimately, a certain amount of change occurs when reviewing newly built systems even when an on-site customer is present.

"As larger, agile projects require teams of teams, communication between the teams becomes especially important. Representing the needs of the solution as a whole is the architects' responsibility."

Working Software Over Documentation[3]

The goal of each iteration is to produce working software. The agile community believes that those activities that do not contribute to this working software are considered lower priority, if not detracting. Unfortunately, there is also a general belief that many of the traditional architectural activities fall into this category.

To be clear, the agile philosophy does not hold a belief that architecture is unbeneficial. Instead, it is a reaction to some of the large design efforts that were performed at the beginnings of projects and later found to be flawed. This form of design is known as *big design up front* (BDUF). The objection that the agile community has to BDUF is that without working software, these efforts have no feedback mechanism. Therefore, quite a bit of time can go into these efforts without an understanding of whether they are constructive or not. Many projects found that their implementation technology did not support these designs and a considerable amount of time had been wasted.

Architecture is an important part of any project, agile or otherwise. It is especially important in the larger agile projects [9]. However, architecture must lead as well as reflect the structure and logic of the working code. Disconnected architectural efforts are often greeted with skepticism by the developers who are building the pieces of the system. However, understanding every detail of a system, especially a larger one is beyond most people's capability. Architects have their hands full just keeping abreast of the changes for iteration. Therefore, keeping the architecture synchronized should be as simple as a whiteboard drawing and as equally expressive [10].

Architects must therefore take a broad view of the system in addition to understanding a certain depth. This breadth is important on larger, more complex projects. When a project spans multiple teams, it is important to communicate responsibility and overall system structure. As larger, agile projects require *teams of teams*, communication between the teams becomes especially important. Representing the needs of the solution as a whole is the architects' responsibility.

To create an agile architecture, MSF utilizes *shadowing*. A shadow is architecture for the functionality to be completed in the iteration. The shadow leads the working code at the beginning of iteration as the architects get out in front of the development for the iteration. During this time, the architecture and the working code are not in sync. This shadow communicates any re-architecting or redesign that needs to occur to keep the code base from becoming a stove pipe, spaghetti code, or one of the many other architectural anti-patterns [11].

As the pieces of the leading shadow are implemented, the architecture begins to reflect the working code base. The original parts of the system that were

architected but not implemented now become implemented. When the architecture represents the working code, we call the shadow a trailing shadow. As the sun sets on the iteration, the leading shadow should be gone and replaced strictly by trailing shadow. The trailing shadow is an accumulation of the architectures over all the iterations.

To keep architecture from becoming too detailed, we recommend that it be focused at the component and deployment levels. For example, a smart client system for generating budget information may consist of a Windows client and a number of Web Services [12]. Each of these Web services, the underlying database server, and the client itself would be components in this model. Remaining at the component level keeps architects from becoming the police of low-level design, although it never hurts to get tips from a more experienced developer.

The Microsoft terminology for one of these deployable components such as a Web service or database server is an *application*. One of the chief tools for the MSF architect is the application diagram, the equivalent of the component diagram in the Unified Modeling Language. Since the application diagram focuses on more concrete entities such as a Windows application, ASP.NET Web service, or external database, more system-level detail can be provided.

Shadowing is applied at the component or applications level. A shadow application initially communicates a desired change in the component-level behavior of a system. Shadow applications become invaluable when multiple teams are trying to coordinate work across multiple components. Changes can be made without affecting the code base until the architecture is ready to be implemented. Next, the code is generated[4] or written for the shadow and the leading shadow is removed and replaced with a trailing shadow.

The planning process for creating shadow applications is similar to the agile pattern used to partition and plan the development work for the system. New architecture tasks are created at the beginning of the iteration when any structural changes need to be made to the architecture to accommodate the new scenarios or quality-of-service requirements. Architecture tasks are like the development or coding tasks that are used to divide the scenarios into the lower-level pieces that can be assigned to a single developer. However, they pertain to the architectural functions that must be performed to keep the system from entropy[5].

As a result of these tasks, the architect will add the endpoints or interfaces to the shadow applications to reflect the needs of the new requirements. These endpoints can be validated to ensure that the components such as Web services will work together properly in the context of the deployment environment. The endpoints of these applications can be connected to show how the components interact. Each application may be distributed on a separate machine or clustered to work together on a single machine.

As the development team becomes ready to implement the scenarios, the endpoints are deleted from the shadow applications and added to the application that represents working code. Unit tests are created for each side of the component to ensure that the proper functionality is provided and unit-tested. Finally, working code is written for these new endpoints.

At the end of the iteration, all of the proxy or unimplemented endpoints should be gone. In other words, all of the architecture should be translated into working code. The architectural model is not divorced from the working system, but rather is a reflection of it. This makes the documentation for the component model match the working system. Unit tests should be in place to make sure that the interfaces continue to work as new functionality is added.

Shadow applications provide many advantages. They keep the high-level design of the components in the system consistent with the code base. They allow larger teams to define responsibilities in the context of an agile architecture. Shadow applications are used to track the building of functionality across component boundaries. In this way, they allow MSF for Agile Software Development to scale to larger, more complex projects.

Individuals and Interactions Over Processes and Tools[6]

The idea of valuing people over processes and tools is not an indication to move away from the use of today's advanced tools. In fact, one of the roots of the agile revolution is the advance of the compiler technology provided by our software development tools. These compilers have made it easier for us to build systems incrementally. If compilation times took hours, as they did in the past, instead of seconds or minutes, can you imagine performing one refactoring at a time? Can you imagine running a unit test first to see it purposely fail after waiting two hours for it to compile?

As our development tools have advanced, so has our capability to take advantage of these advances in our development processes. However, developing software is ultimately an activity for knowledge workers. The static nature of tools and processes is no match for the adaptation that people can provide to deal with the ever-changing nature of our project and our industry.

Each project operates under a different climate and set of working conditions. The factors that influence a project include size, criticality, deadline, and required quality. There is a general perception that you always need to change the process to deal with these project differences. Creating agile processes for each of these types of projects would mean that there would be hundreds of new agile processes. Instead, we can understand how these factors affect our process and utilize a context-based approach.

A context-based process allows us to tune the process to the context of our project. The quality criteria used for release are often driven by the project type. Context-driven testing bases the test approach on the factors of each project as well. The idea behind context-driven testing is that the successful approach to testing one type of application may be criminal on another type. Test thresholds, metrics for determining the shipping quality, may be used to govern the test and release approach.

The test thresholds are determined by the project team and recorded by the test team. Only one test threshold is required of an MSF project. This is code coverage for unit tests, a metric that measures the percentage of code that is tested by a set of unit tests. Like many of the other agile processes, MSF for Agile Software Development requires unit testing as part of its development activities.

However, the effectiveness of this safety net is measured in MSF. Normal test-driven development can account for 50 percent to 70 percent code coverage on many projects, but to achieve higher levels of code coverage requires more complex techniques such as mock objects [13]. Some projects, like a data converter for a one-time use, may be fine with a lower unit testing code coverage threshold for this safety net. A critical system such as an automatic pilot system probably requires a greater level of unit testing.

These metrics may be extended to

govern the project as well. For example, maximum bug debt, the maximum number of bugs that a developer may have, can be used to determine when an iteration devoted to fixing bugs (called bug allotment iteration) should be scheduled. When the number of bugs exceeds this threshold, this is an indication for the project manager to provide a whole or part of iteration for fixing bugs.

Responding to Change Over Following a Plan[7]

Wouldn't it be nice if you knew exactly what had to be done at the beginning of a project? How about if there were absolutely no surprises during the project? There are a few very small, straightforward projects that enjoy this nirvana. When the rest of us try to achieve this ideal condition, we find ourselves faking a rational design process or behaving as if change does not happen [14].

However, in the real world of software development, requirements change. We may also discover an aspect of the technology that we are using that we did not previously know. We learn about the system that we are building in the process of building it. The fact is, we can talk about these fairy-tale projects where change does not occur, but reality has a nasty habit of creeping in.

So why not plan for reality rather than trying to aspire to a mythical standard that is unattainable? In fact, we can do even better; we can use reality to develop more optimal software development processes. While our business analysts are gathering the requirements, what are our developers doing? How about our project managers? While our project managers are planning, what are our developers doing? How about our testers?

The answer is that they should all be working in parallel. While our business analysts understand the requirements, our project managers are planning, our developers are developing, and our testers are testing. How can we do this? We accomplish this through staggering the work, setting up coordination points, and providing only what is needed in a just-in-time fashion. For example, we only write the scenarios for the upcoming iteration, we plan one iteration at a time, architect only the necessary pieces, develop the functionality in the iteration plan only for this iteration, and write test cases for functionality planned in the current iteration.

Conclusion

Personas, shadow applications, and test thresholds are part of Microsoft's new agile software development process, MSF for Agile Software Development. These techniques provide alternate ways to satisfy the value statements of the Agile Alliance. They have been proven through their repeated use in delivering Microsoft software development projects.

Becoming agile is as much about changing your state of mind as it is the adoption of new practices. This article shows some new techniques to introduce agile software development to many of the roles that have not been included in many of the agile processes. By using these techniques in an agile way, we can extend agile software development processes to the entire organization.◆

References

1. McMahon, Paul E. "Extending Agile Methods: A Distributed Project and Organizational Improvement Perspective." CrossTalk May 2005 <www.stsc.hill.af.mil/crosstalk/2005/05/0505McMahon.html>.
2. Microsoft Developer Network. The MSF for Agile Software Development Workbench. Microsoft Corporation, 18 Sept. 2005 <http://lab.msdn.microsoft.com/teamsystem/workshop/msfagile/default.aspx>.
3. Miller, Granville. "Want a Better Software Development Process? Complement It." IEEE IT Professional 5.5 (Sept./Oct. 2003): 49-51.
4. Beck, Kent, and Martin Fowler. Planning Extreme Programming. Addison-Wesley, 2000.
5. Schwaber, Ken. Agile Project Management With Scrum. Microsoft Press, 2004.
6. Beck, Kent, et al. "Manifesto for Agile Software Development." Agile Alliance, Feb. 2001 <www.agilemanifesto.org>.
7. Cooper, Alan. The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity. 2nd ed. Sams, 2004.
8. Armour, Frank, and Granville Miller. Advanced Use Case Modeling: Software Systems. Addison-Wesley, 2000.
9. Eckstein, Jutta. Agile Software Development in the Large: Diving Into the Deep. Dorset House Publishing, 2004.
10. Ambler, Scott. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. Wiley, 2002.
11. Brown, William J., Raphael C. Malveau, Hays W. "Skip" McCormick (III), Thomas J. Mowbray. Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1998.
12. Boulter, Mark. Smart Client Architecture and Design Guide. Microsoft Press, 2004.
13. Astels, David. Test Driven Development: A Practical Guide. Prentice Hall, 2003.
14. Parnas, David Lorge, and Paul C. Clements. "A Rational Design Process and How to Fake It." IEEE Transactions on Software Engineering 12.2 (Feb. 1986): 251-257.

Notes

1. This is the third value statement from the agile manifesto [6].
2. There are many different names for this role depending on whether the project is created for commercial or internal use. The role name is not as important as the function that it performs.
3. This is the second value statement from the agile manifesto [6].
4. Class or method structure for the high-level components can be generated from a shadow.
5. Entropy is the tendency for software to become brittle and difficult to add to or change over time.
6. This is the first value statement from the agile manifesto [6].
7. This is the fourth value statement from the agile manifesto [6].

About the Author

Granville "Randy" Miller is the architect of Microsoft's agile software development process, Microsoft Solutions Framework for Agile Software Development. He has two decades of experience in the commercial software industry and has spoken at many international events, including XP200x, Conference On Object Oriented Programming Systems, Languages and Applications, Web Services Edge, Software Development West, Microsoft TechEd, and others. His interests include software development technology and agile software development. Miller is author of "Advanced Use Case Modeling" and "A Practical Guide to Extreme Programming."

Microsoft
randymil@microsoft.com

Eliminating Embedded Software Defects Prior to Integration Test

Ted L. Bennett and Paul W. Wennberg
Triakis Corporation

Research has shown that finding software faults early in the development cycle not only improves software assurance, but also reduces software development expense and time. The root causes of the majority of embedded system software defects discovered during hardware integration test have been attributed to errors in understanding and implementing requirements. The independence that typically exists between the system and software development processes provides ample opportunity for the introduction of these types of faults. This article shows a viable method of verifying object software using the same tests created to verify the system design from which the software was developed. After passing the same tests used to verify the system design, it can be said that the software has correctly implemented all of the known and tested system requirements. This method enables the discovery of functional faults prior to the integration test phase of a project.

New, complex embedded systems are quick to take advantage of the unrelenting pace of advancement in computer hardware performance and capacity. Along with the increase in hardware capability comes a considerably greater increase in the functionality and complexity of the software in control.

Unfortunately, the methods and tools we use to develop and test systems and software have not kept up with the trend. This is evidenced by the number of software faults that pass undetected into the integration and operational phases of contemporary projects.

This is of concern for two important reasons. In the case of software in control of safety – or mission-critical systems – allowing a failure to pass undetected into the operational phase of a project may put lives and/or critical missions at risk. In all cases, the more faults that pass undetected into integration test and beyond, the more the project will cost and the longer it will take to complete.

This article presents a new, closed-loop method of simulating and verifying embedded system designs and their controlling software in a pure, virtual system integration laboratory environment. We have demonstrated and validated this method in a recently concluded research effort sponsored by the NASA Office of Safety and Mission Assurance under their Software Assurance Research Program [1]. Our investigation showed the following:

1. A new method of specifying, executing, and verifying an entire system design in a pure virtual environment.
2. How uninstrumented, embedded object software can be verified in the virtual system environment.
3. How the same tests used to verify the system design may be used to verify the controlling software.

   It follows from item No. 3 that if the software passes the same tests used to verify the system design then it correctly implements the known and tested system requirements. As a result, we now have a viable means of discovering requirements-induced software faults prior to the integration test phase of a project. This is significant because it has been shown that early discovery of faults reduces both project cost and duration.

Root Causes of Software Faults

The root causes of the majority of software defects discovered in integration test during the development of an embedded system have been attributed to errors in understanding and implementing requirements (see the sidebar "JPL Root-Cause Analysis of Spacecraft Software Defects" and Figure 1 on page 14). These may be the system and/or the software requirements. We assert that this is largely a result of the independence that exists between the requirements development and the software development processes.

The JPL report findings are echoed in reports of numerous other researchers such as Leveson [3, 4], Ellis [5], Thompson [6], and others. Consider some of the many avenues where requirements-related problems might be introduced:

- Assumptions/ambiguities affecting the interpretation of customer descriptions of desired system behavior.
- The difficulty in fully understanding the real-world environment in which the system will interact.
- The difficulty in anticipating all of the possible modes and states that the system may encounter.
- The difficulty in thoroughly validating and verifying requirements.
- Capturing accurate, unambiguous representations of requirements in a written document.
- Misinterpretation of system-level requirements by software designers.
- The difficulty in verifying that the design has correctly implemented the requirements.

To compound the problem, we generally cannot know at the onset of a project if we have accurately modeled the real-world system behavior. As a project advances, however, so does our understanding of the system. Additional faults may be introduced when subsequent refinements to the system model are not adequately communicated to the software development teams. To be more effective at creating software with a high level of assurance, not only must we reduce the number of errors attributable to misunderstanding and misimplementing requirements, but we must also improve communication between and among the system and implementation teams.

Shortcomings of Federated Development Methods

Contemporary, embedded systems development projects are typically conducted in a federated manner. In other words, the system and software development activities are conducted essentially independent of each other. To illustrate this point, Figure 2 on page 15 depicts the three principal loops comprising a typical project process. We will ignore hardware development activities since they are not germane to this discussion.

The first loop is where the system design is created. The system designers may make use of modeling, simulation, prototyping, executable specifications (ES), and other tools to satisfy the need to validate control algorithms, component interactions, etc. The system architects validate and verify their design through analysis, possibly tests, and possibly by similarity with reused components. They

JPL Root-Cause Analysis of Spacecraft Software Defects

In 1992, Dr. Robyn Lutz conducted an analysis for the Jet Propulsion Laboratory (JPL) to determine the root causes of the 387 software defects discovered during the integration test phase of the Voyager and Galileo spacecraft development efforts. The software controlling these spacecraft is distributed among several embedded computers with roughly 18,000 and 22,000 lines of source code, respectively. Lutz reported that the programming faults discovered on the two projects are distributed as shown in Figure 1.

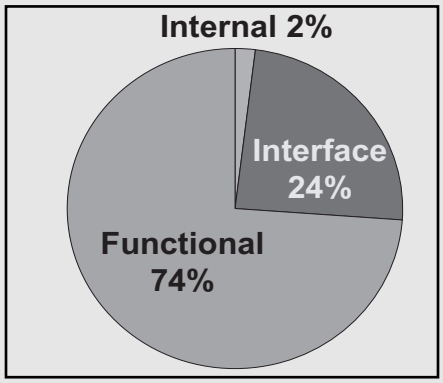


Figure 1: *Fault Distribution*

The fault classifications given in Figure 1 are defined as follows:

- *Functional faults* comprise the three subclasses listed below:
  a. Operating faults: Omission of, or unnecessary operations.
  b. Conditional faults: Incorrect condition or limit values.
  c. Behavioral faults: Incorrect behavior, not conforming to requirements.
- *Interface faults* are those related to interactions with other system components such as transfer of data or control.
- *Internal faults* are defined as coding faults internal to a software module.

The data show that 98 percent of the combined total software problems were classified as functional or interface faults that are directly attributable to errors in understanding and implementing requirements, and inadequate communication between development teams. Only 2 percent were due to software module coding errors [2]. The conclusions of the JPL report point to the need for improved focus in the following areas:

1. Interfaces between the software and the system domains.
2. Identification of safety-critical hazards early in the requirements analysis.
3. Use of formal (and unambiguous) specification techniques.
4. Promotion of informal communication among teams.
5. Keeping development and test teams apprised of changes to requirements.
6. Inclusion of requirements for *defensive design*.

then document the requirements for the implementation teams to follow. When satisfied with their design (or when time runs out), the system team delivers the system specification package to the implementation teams.

Entering the second loop shown in Figure 2, the software implementation team interprets the relevant requirements – whether written in natural language, specification design language, or executable specifications – derives software requirements, and creates its design. The software developers write their own tests to verify conformance to the requirements as they have interpreted them. They may use some form of simulation, hardware development boards, inspection, analysis, or similarity comparison to facilitate verification of their code.

When a major part of the system functionality has been coded, the software team creates a build. The software is loaded into its target hardware where integration test begins in the laboratory. Connected to test equipment, simulators, and perhaps other system elements, the control software is stimulated by the hardware environment under the control of custom test software. Bugs discovered during integration test are filed as problem reports and passed back to the development team to resolve, thereby completing the third loop.

We see the independence that exists between the system and software loops in this development process as the primary reason for the propagation of software faults into integration test. Further, this independent process may breed duplicity of effort where the software and system teams write their own tests to verify the same behavior at the system and software levels.

Our research has shown a method of connecting the system and software development loops that allows tests written for system verification to be used to verify the software itself. This enables the software to be thoroughly debugged in a pure, virtual environment before it ever gets to the hardware integration phase.

Coupling the System and Software Development Loops

Figure 3 illustrates our approach to connecting the system and software development loops. This new approach retains the system and software development loops, but eliminates the loop where the hardware integration lab is used for software debug activities.

As before, your project begins with the development of a system design using various tools for algorithm development, etc. However, in lieu of passing the design and requirements to the implementation teams as a collection of disparate specifications, the entire system and the environment in which it interacts is simulated using a form of ES. All parts in the simulation are bounded like their real-world counterparts so the interface behavior of each element can be correctly modeled and specified. Parts are created with built-in failure modes that may be activated under test control.

Having modeled the behavior of the entire system environment, you now have a complete virtual system integration laboratory (VSIL) in which to validate and verify your system design. The next step is to create a suite of tests based upon nominal and off-nominal scenarios for which the system has been designed to react. Our testing philosophy is to exercise the system by driving the environment as realistically as possible, and monitoring the system behavior in response. This is generally not a viable approach for hardware system integration laboratory setups due to the cost or difficulty involved in procuring, creating, and synchronously controlling all the disparate pieces of hardware and simulators necessary to realistically drive the target system.

The completed and verified VSIL is then passed, along with the system-level tests and any supplemental written requirements, to the development teams. The teams create hardware and software designs from the specified processing, communication, interface, control, and other requirements. As soon as the hardware architecture has been established, the target embedded controller for which the software is being developed must be simulated with sufficient fidelity to run the unmodified object software. Because the simulated controller hardware is bounded (i.e., it has identical interfaces) like the ES part from which it was developed, it may be plugged into the VSIL in

place of its ES counterpart. We refer to this controller hardware simulation part as a detailed executable (DE) (see Figure 3).

The DE gives the software team the ability to test the software it develops (see Figure 3, step 1) in the VSIL (see Figure 3, steps 2-4). After replacing the controller ES with the DE, the software being developed may be compiled and loaded into the DE at any time for testing in the VSIL. All of the tests created to verify the system design can be used, without modification, for software verification. Additional tests must be added to verify that software has correctly implemented lower-level requirements whose detail has not been addressed at the system level (e.g., built-in test, etc.).

After running the desired tests, the software development team analyzes the results and determines the cause of any failures. The team then corrects any identified faults, recompiles the revised modules, and retests the build in the VSIL (see Figure 3, steps 1-4). In practice, step 3 is performed once since the DE becomes an integral part of the VSIL following replacement of its ES counterpart. The VSIL is tightly coupled with the integrated software development environment used by the software team – thereby facilitating the code/compile/load/verify process.

Some of the problems discovered may require the attention of the system designers. When this necessitates a system design change, the VSIL is revised and tested and redistributed to the software development teams. In this manner, the software is always developed and tested in the most current system design – thereby eliminating the possibility of software problems being introduced due to miscommunication of system design changes.

The software design/code/verify/debug loop is repeatedly executed until the final build passes all tests and until all paths through the code have been exercised in the VSIL. Thus, the software has been thoroughly verified and is ready for integration testing with the real flight hardware.

It is worth noting that since the object code itself is tested in the VSIL, the real-time operating system (RTOS), any reused/commercial off-the-shelf modules, and all newly developed software are verified together in the virtual target environment. The VSIL itself is a Microsoft Windows-compatible application that interfaces with standard integrated development environment tools. A VSIL is as easily used as a typical lab test setup (e.g., emulator, simulators, target hardware) and readily distributed to all project development personnel. Since the entire system and environment are modeled in the VSIL, modifications and refinements can be coded, validated, verified, and distributed to the entire team. VSIL revisions and verification tests may be controlled using standard configuration management tools and techniques. Lastly, the VSIL is purely virtual: no hardware is required other than the Windows-based PC on which it runs.

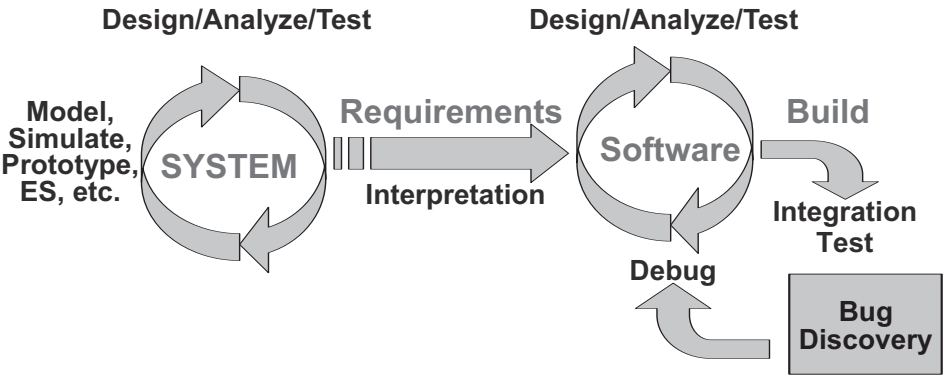


Figure 2: *Federated Development Process*

Discussion

We have presented a new method of embedded systems and software verification and validation (V&V) that closes the loop between system and software development activities. In so doing, the system and software development processes can now be connected through common verification tests.

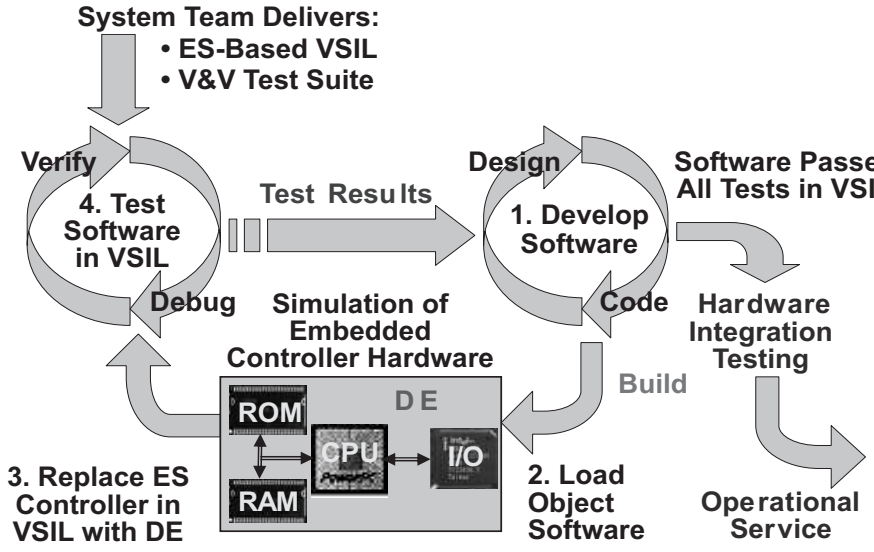
Finding and repairing software faults early in the project development cycle can lead to substantial savings (see the sidebar "Economics of Fault Finding" on page 16). For example, requirements and communication-induced errors like 98 percent of those discovered during the integration phase of the Voyager and Galileo spacecraft software projects, can be found and repaired at one or perhaps more orders of magnitude lower cost.

Implications

Below is a summary list of some of the ways that the methods presented in this article may be of economic benefit to embedded software development:

1. Discover system errors early in the development cycle where it is least costly to correct them.
2. Reduce interpretation-induced software faults due to ambiguities in system requirements.
3. Improve ability for dynamic, non-invasive test of system and software response to failure conditions.
4. Reduce software faults caused by breakdown in communication of system requirements changes.
5. Utilize new capacity for empirical software V&V in cases where analysis was the only viable means, for example, realistic fault injection and failure mode testing, complex digital signal processor designs, etc.
6. Provide a highly viable means of verifying automatically generated code,

Figure 3: *Closed-Loop Software Verification in Virtual System Integration Lab*



Economics of Fault Finding

Estimates of the cost to find and correct software faults at each of the principal stages of a project have been publicized and widely referenced since 1976 when Boehm first published his study [7] on the subject. Cost numbers vary depending on the type of application for which the software is being developed, but the common thread they all exhibit is the substantial increase in project costs caused by carrying problems from one development stage to the next.

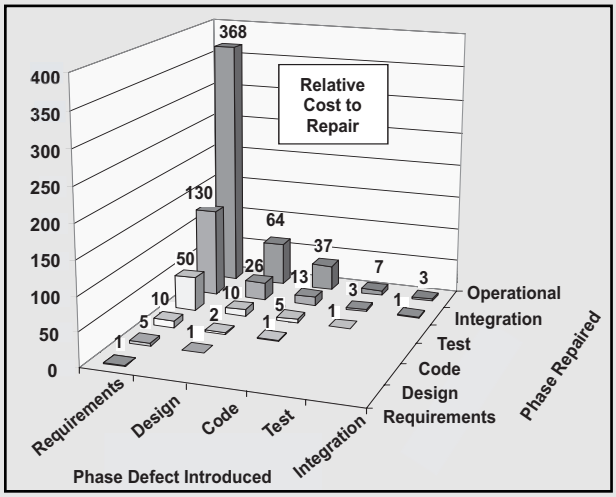
A report released in May 2002 by the National Institute of Standards and Technology (NIST) [8] contains a thorough analysis concluding that inadequate software testing costs the United States an estimated *$59.5 billion annually*. The 309-page NIST report is a well-considered treatise on the economic impact of inadequate software testing.

While these numbers are extrapolated from software developed for the financial services and transportation applications (computer-aided design, computer-aided manufacturing, etc.) sectors, the message applies even more significantly to industries engaged in developing software for safety and mission-critical applications such as aerospace, medical, defense, automotive, etc. Failures of safety/mission-critical software may result in harm to, or loss of human life and/or mission objectives such as in the case of the Therac-25 radiation overdose accidents [2] and the Ariane-5 maiden launch failure [9]. The Therac-25 software caused severe radiation burns in numerous cancer patients before it was implicated. The cost of allowing the Ariane-5 software defect to pass into the operational phase has been estimated to be as high as $5 billion alone.

NASA recently sponsored a study to evaluate the economic benefit of conducting independent validation and verification during the development of safety-critical embedded systems [10]. This study presented cost-to-repair figures focused specifically on embedded systems projects. Figure 4 shows the relative cost to repair factors – considered to be conservative estimates for embedded systems – used in this study.

The graph in Figure 4 tells us that an error introduced in the requirements phase will cost five times more to correct in the design phase than in the phase in which it was introduced. Correspondingly, it will cost 10 times more to repair in the code phase, 50 times more in the test phase, 130 times more in the integration phase, and 368 times more when repaired during the operational phase. The graph also gives the cost multipliers for problems introduced in the design, code, test, and integration phases of the development cycle.

Figure 4: *Relative Cost of Software Fault Propagation*



reused software, and RTOS.

Creating a system design with the type of ES discussed herein results in a verifiable system architecture that is readily translated into component- and interface-level designs. When contracting out the development of subsystem software, the system-level verification tests can provide an excellent way to assure that the contractor has developed the software correctly.

Because ES parts may be created with intrinsic failure modes that can be invoked dynamically under test control, the system designer can empirically verify the specified system response to a variety of off-nominal conditions. This ability allows greater latitude in the type and number of tests that can be conducted when compared with what is economically viable in a hardware integration lab.

Verifying the VSIL

The VSIL is, in fact, a model of both the system being developed and the environment in which it is designed to interact. Before it can be of use, we must have confidence that the VSIL represents its target adequately.

We have adopted an effective approach that is perhaps best described as test as you go. As parts are simulated to implement specific requirements, system-level tests are created simultaneously to verify that they behave correctly. Part functionality may be developed and tested incrementally as requirements are implemented. At the end of this process, all VSIL parts have been implemented and verified and a basic set of system-level tests has been developed.

Parts developed to a high fidelity level may require a supplemental verification activity where the real-world equivalent part is used for comparison purposes. In the case of developing an instruction-set-level CPU simulation, we run test code designed to verify instruction execution on a hardware development board and compare the results with the outcome of running the same code on the simulated part. The CPU parts we have developed are not cycle-accurate but are refined to where the instructions execute within an average of 4 percent of the hardware performance (works well for embedded software verification). This is in keeping with our philosophy of not implementing greater fidelity than necessary.

VSIL Development Tool

Triakis developed its first avionics simulator more than a decade ago to save time verifying software modifications and to avoid contention for lab test resources. This initiative spawned the creation of IcoSim, Triakis' general-purpose simulator development tool, and its companion software developer's kit (SDK).

The IcoSim SDK is typically available at no cost to customers availing themselves of Triakis' VSIL development services. In the second quarter of 2006, however, Triakis plans to make IcoSim freely available to the general public by turning IcoSim into an open source project[1] whose use will be governed under a Lesser General Public License (LGPL) [11]; simulated parts will be covered individually under a LGPL, GPL [12], or proprietary license.

Tool Description

Since it is destined to become an open source project, the descriptive details provided herein are intended to promote an understanding of how we accomplish what we have presented.

Written in C++ and C, IcoSim allows the use of diverse part types ranging from low to high abstraction levels. It also supports using mixed mode parts such as analog, digital, mechanical, hydraulic, magnetic, electromagnetic, et al.

IcoSim is well suited to creating a VSIL for use in developing embedded

systems and software because the simulated parts may be bounded exactly like their real-world counterparts. In other words, the inputs and outputs of each virtual part are readily modeled after the behavior of their real-world part's digital, analog, mechanical, etc. input/output. Once its behavior is verified, a virtual part may be identified with the same part number as its counterpart, and repeatedly used wherever system designs specify.

VSIL Parts Libraries

In addition to the NASA research that validated the methodology presented, IcoSim has been used to create VSILs for software verification on more than two dozen avionics projects over the past decade. It is scalable to any size system and has been used for verification of software in single and dual-redundant avionics systems ranging in criticality from Radio Technical Commission for Aeronautics, Inc. (RTCA) Defense Order (DO)-178B[2], level A (safety-critical) to level D (low criticality). It has also been used for verification of embedded digital signal processor (DSP) software implementing Kalman filter algorithms.

Triakis' parts library includes instruction-set-level simulations of many microprocessors in use today such as the MPC555, MPC750, RAD6000, MC68000, MC68332, DSP56005, DSP56302, DSP56309, I80196, I8051, I8096, I8097, I8798, et al. Numerous additional peripheral and glue parts are in the library as well as a host of actuators and sensors that have been created in support of various VSIL projects. Triakis has also created a collection of parts that simulate many different data buses and protocols, e.g., Aeronautical Radio, Inc. (ARINC) 419; ARINC 739; Military-Standard-1553; Time-Triggered Protocol; Avionics Standard Communication Bus; Commercial Standard Data Bus; Avionics Full-Duplex Switched Ethernet; Serial Peripheral Interface; Peripheral Component Interconnect, Controller Area Network, etc.

To support testing with a VSIL, we have simulated standard laboratory test equipment such as oscilloscopes, signal generators, and the functional capability of microprocessor emulators. The VSIL is an ideal environment for gathering dynamic software metrics without instrumenting either the target operating system or the software. Code path coverage, Modified Code Decision Coverage reports, throughput analysis, timing analysis, and many other helpful reports are readily produced in this environment with the addition of instructions to the test script.

Costs of VSIL Development

A VSIL is made by interconnecting objects at the lowest level of abstraction to make successively higher levels of functional parts until the required environment is complete. This hierarchical, modular approach maximizes the potential for part reuse on subsequent development projects.

To be efficient at making a VSIL, each part is simulated only to the level of fidelity necessary to achieve one's goals. For example, an aircraft rudder is attached to a sensor that reports its angular position to avionics subsystems as required. The sensor has a mechanical link to the rudder, has inertial properties, may have inductive coils, may have an armature, may be excited by a 400 Hz reference, etc.

"There are many factors that influence the cost, but a typical VSIL [virtual system integration laboratory] can be developed for about 5 percent to 10 percent of the overall project cost."

While we could model all of these characteristics with great precision, it would be a waste of effort if our system only required the correct transfer function of rudder angle to sensor output at a given update rate. Since part fidelity is directly proportional to effort, being selective about where to incorporate higher fidelity is key to cost-effective VSIL creation.

It is difficult to quantify the costs of creating a VSIL for system and software development because of the large number of variables involved such as the following:

- System size.
- System complexity.
- Number of parts to be simulated.
- Number of control processing units to be simulated.
- Experience of simulation engineer(s).

Because of the part-oriented nature of the VSIL, the cost of creating a simulator for a given project will vary in proportion to the number and complexity of new parts that must be created. Many new, embedded designs reuse proven design elements from prior projects so the cost of developing simulators diminishes with successive applications.

Supplemental VSIL Benefits

The benefit of using a VSIL for embedded systems and software development increases with project size, system complexity, and geographic diversity of organizations and personnel contributing to the project.

In addition to the cost benefits of early software fault discovery, a VSIL can support a project in other important ways. Some of these benefits are directly measurable, but others may have less tangible value:

- When contracting out development of a subsystem, supplying the vendor with a VSIL and its system test suite can be a highly effective means of verifying that the software conforms to the requirements.
- Development teams in local and remote locations can quickly re-verify their software following system revisions that have been implemented and tested in a VSIL. Using standard configuration control procedures, the latest system revision can be distributed to all teams as soon as it is available.
- Providing a VSIL to every programmer promotes a broader, big-picture understanding of the system. Every programmer tests on the whole system, every time.
- Testing in a VSIL reduces the dependence on laboratory test stations; consequently, fewer are required.
- Less dependence on laboratory test equipment reduces resource-contention delays during development.
- A VSIL may be helpful in the operational phase of a project for the following:
  - Software re-verification following upgrade modifications with full regression testing.
  - Re-verifying software on obsolescent-driven hardware design changes.
  - Verification of system compatibility with upgrades to peripheral or subsystem units.
  - Eliminating or reducing reliance on test equipment setups that must be maintained to support software changes following entry into service.

While not a rigorous analysis, one avionics company's post-project review of having used a VSIL for verification of their dual-redundant avionics software revealed some attractive cost-benefits.

Based on their findings they concluded that future projects could expect a 24 percent schedule savings, a $130,000 direct savings on laboratory equipment, and realize an overall cost savings of 14 percent on an average $4.5-million project. These estimates do not take into account the benefits afforded by a VSIL throughout the operational life of a product. There are many factors that influence the cost, but a typical VSIL can be developed for about 5 percent to 10 percent of the overall project cost. This places the return on investment in the range of 40 percent to 180 percent for the above project.

Experiences will no doubt vary from project to project; however, these estimates can provide useful guidance when assessing the life-cycle cost/benefit of using a VSIL for development.

Summary

The new method of embedded systems and software V&V presented here goes far beyond an incremental improvement to the status quo. While not a panacea, it does provide a cost-effective, proven means of the following:

- Ensuring that the target software has implemented all known and tested system requirements – prior to hardware integration.
- Verifying automatically generated code, reused software, and the RTOS.
- Verifying response of systems and software to a wide range of realistic, dynamic failures and off-nominal scenarios.
- Re-verifying software following system revisions and updates.
- Ensuring that hardware redesigned for obsolescence is compatible with the software.
- Verifying that new and upgraded peripherals and subsystems function correctly with the target system.

The approach described provides a bridge between algorithm and model development tools, and the real-world system environment in which embedded algorithms must function. This method is a highly viable way to address a number of problems that hamper efficient embedded systems and software development.◆

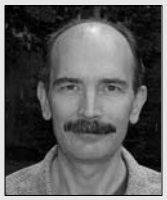
References

1. Bennett, T.L., P.W. Wennberg. "The Use of a Virtual System Simulator and Executable Specifications to Enhance Software Validation, Verification, and Safety Assurance – Final Report." Software Assurance Research Program Results Web Site. Fairmont, West Virginia : NASA IV&V Facility, June 2004. <http://sarpresults.ivv.nasa.gov/ViewResearch/285/32.jsp>.
2. Lutz, R.R. "Analyzing Software Errors in Safety-Critical, Embedded Systems." Pasadena, CA: Jet Propulsion Laboratory, California Institute of Technology, 1994.
3. Leveson, N.G. Safeware – System, Safety, and Computers. Addison-Wesley, 1995.
4. Leveson, N.G. "Software Safety: What, Why, and How." ACM Computing Surveys 18.2 (1986).
5. Ellis, A. Achieving Safety in Complex Control Systems. Proc. of the Safety-Critical Systems Symposium. Brighton, England: Springer-Verlag, 1995: 2-14.
6. Thompson, J.M. "A Framework for Static Analysis and Simulation of System-Level Inter-Component Communication." Masters Thesis. University of Minnesota, 1999.
7. Boehm, B.W. "Software Engineering." IEEE Transactions on Computer 1.4 (1976): 1226-1241.
8. Tassey, G. "The Economic Impacts of Inadequate Infrastructure for Software Testing." National Institute of Standards and Technology, 2002 <www.nist.gov/director/progofc/report02-3.pdf>.
9. Leveson, N.G. "The Role of Software in Spacecraft Accidents." AIAA Journal of Spacecraft and Rockets 41.4 (July 2004).
10. Dabney, J.B. "Return on Investment of Independent Verification and Validation Study Preliminary Phase 2B Report." Fairmont, W.V.: NASA IV&V Facility, 2003. <http://sarpresults.ivv.nasa.gov/ViewResearch/289/24.jsp>.
11. GNU Lesser General Public License. Vers. 2.1. Boston, MA: Free Software Foundation, Inc., 1999 <www.opensource.org/licenses/lgpl-license.php>.
12. Open Source. The General Public License (GPL). Vers. 2. Boston, MA: Free Software Foundation, Inc., 1991 <www.opensource.org/licenses/lgpl-license.php>.

Notes

1. Details about open-source projects can be found at <http://sourceforge.net/>.
2. Information about DO-178B can be found at <www.software.org/quagmire/descriptions/-178b.asp>.

About the Authors



Ted L. Bennett is director of Systems Engineering and Business Development at Triakis Corporation. He has more than 25 years experience in embedded hardware and software design, systems engineering, project management, and business development in the aerospace industry. Bennett was principal investigator for the NASA-sponsored research project that validated the breakthrough methodology presented in this article. He is also principal investigator on two additional NASA research grants currently being conducted by Triakis. He has a Bachelor of Science in electrical engineering from the University of Wisconsin at Madison.

Triakis Corporation
16149 Redmond WY STE 177
Redmond, WA 98052
Phone: (425) 558-4241
Fax: (425) 558-7650
E-mail: ted.bennett@triakis.com

Paul W. Wennberg is president and founder of Triakis Corporation and conceived and created IcoSim, the pure virtual environment simulator tool discussed in this article. He has over 20 years experience in the design and test of embedded systems hardware and software, and pioneered this new methodology. A U.S. Air Force veteran, Wennberg logged over 1,400 hours piloting T38 and KC135 aircraft prior to completing his service with the rank of captain. He has a Bachelor of Science in electrical engineering from the University of Washington at Seattle.

Triakis Corporation
16149 Redmond WY STE 177
Redmond, WA 98052
Phone: (425) 861-3860
Fax: (425) 558-7650
E-mail: paul.wennberg@triakis.com

Acquiring Quality Software

Watts S. Humphrey
Software Engineering Institute

If you do not insist on getting quality software, you probably will not get it! That is the first principle of software quality. To get quality software at reasonable costs and on predictable schedules, you must follow the six principles of software quality. This article describes these principles and discusses how to apply them in software acquisition.

In today's software marketplace, the principal focus is on cost, schedule, and function; quality is lost in the noise. This is unfortunate since poor quality performance is the root cause of most software cost and schedule problems. However, as this article points out, there are proven ways to address this problem. The first step is adopting and demanding that vendors follow these six principles of software quality:

- **Quality Principle No. 1:** If a customer does not demand a quality product, he or she will probably not get one.
- **Quality Principle No. 2:** To consistently produce quality products, the developers must manage the quality of their work.
- **Quality Principle No. 3:** To manage product quality, the developers must measure quality.
- **Quality Principle No. 4:** The quality of a product is determined by the quality of the process used to develop it.
- **Quality Principle No. 5:** Since a test removes only a fraction of a product's defects, to get a quality product out of test you must put a quality product into test.
- **Quality Principle No. 6:** Quality products are only produced by motivated professionals who take pride in their work.

These are not just theoretical principles, and almost any software group can follow them, as demonstrated by the experiences of many organizations with the Software Engineering Institute's (SEI[SM]) Team Software Process[SM] (TSP[SM]). All it takes to start down this road is to recognize and act on quality principle No. 1.

Quality Principle No. 1

If the customer does not demand a quality product, he or she will probably not get one.

If you want quality products, you must demand them. But how do you do that? That is the subject of this article. I first define quality, then I discuss quality management, and then third, I cover quality measurement. Next I describe the methods for verifying the quality of software products before you get them, and finally, I give some pointers for those acquisition managers who would like to consider using these methods. That, of course, is the most critical point; even when you demand quality, if you cannot determine that you will get a quality product before you get it, you are no better off than you are today – struggling to recover from the effects of getting poor-quality products.

[SM] SEI is a service mark of Carnegie Mellon University.

> *"In the broadest sense, a quality product is one that is delivered on time, costs what it was committed to cost, and flawlessly performs all of its intended functions."*

Defining Quality

Product developers typically define a quality product as one that satisfies the customer. However, this definition is not of much help to you, the customer. What you need is a definition of quality to guide your acquisition process. To get this, you must define what quality means to you and how you would recognize a quality product if you got one.

In the broadest sense, a quality product is one that is delivered on time, costs what it was committed to cost, and flawlessly performs all of its intended functions. While the first two of these criteria are relatively easy to determine, the third is not. These first two criteria are part of the normal procurement process and typically receive the bulk of the customer's and supplier's attention during a procurement cycle, but the third is generally the source of most acquisition problems. This is because poor product quality is often the reason for a software-intensive system's cost and schedule problems.

Think of it this way: If quality did not matter, you would have to accept whatever quality the supplier provided, and the cost and schedule would be largely determined by the supplier. In simplistic terms, the supplier's strategy would be to supply whatever quality level he felt would get the product accepted and paid for. In fact, even if you had contracted for a specific quality level, as long as you could not verify that quality level prior to delivery and acceptance testing, the supplier's optimum strategy would be to deliver whatever quality level it could get away with as long as it was paid.

Since, at least for software, most quality problems do not show up until well after the end of the normal acquisition cycle, you would be no better off than before. I do not mean to imply that this is how most suppliers behave, but merely that this would be its most economically attractive short-term strategy. In the long term, quality work has always proven to be most economically attractive.

Addressing the Quality Problem

In principle, there are only two ways to address the software quality problem. First, use a supplier that has a sufficiently good record of delivering quality products so you will be comfortable that the products he provides will be of high quality. Then, just leave the supplier alone to do the development work. The second choice would be to closely monitor the development process the supplier uses to be assured that the product being produced will be of the desired quality.

While the first approach would be ideal, and that is the principle behind the successful Capability Maturity Model® Integration evaluation strategy, it is not useful when the supplier has historically had quality problems or where his current performance causes concern. In these

cases, you are left with the second choice: to monitor the development work. To do this, you must consider the second principle of quality management.

Quality Principle No. 2

To produce quality products consistently, developers must manage the quality of their work.

Managing Product Quality

While you may want a quality product, if you have no way to determine the product's quality until after you get it, you will not be able to pressure the supplier to do quality work until it is too late. The best time to influence the product's quality is early in its development cycle where you can determine the quality of the product before it is delivered and influence the way the work is done. At least you can do this if your contract provides you the needed leverage.

This, of course, means that you must anticipate the product's quality before it is delivered, and you must also know what to tell the supplier to do to assure that the delivered product will actually be of high quality. Therefore, the first need is to predict the product's quality before it is built. This is essential, for if you only measure the product's quality after it has been built, it is too late to do anything but fix its defects. This results in a defective product with patches for the known defects. Unless you have an extraordinarily effective test and evaluation system, you will not then know about most of the product's defects before you accept the product and pay the supplier.

While you might still have warranties and other contract provisions to help you recover damages, and you might still be able to require the supplier to fix the product's defects, these contractual provisions cannot protect you from getting a poor quality product. Because most suppliers are adept at avoiding liability for defects, you have not gained very much by contracting for quality. To get the benefits of including quality provisions in your contracts, you must determine the likely quality of the product during development.

Identifying Quality Work

To determine the likely quality of a product while it is being developed, we must consider the third principle of quality work.

Quality Principle No. 3

To manage product quality, the developers must measure quality.

To monitor product quality before delivery you must measure quality during development. Further, you must require that the developers gather quality measurements and supply them to you while they do the development work. What measures do you want, and how would you use them? This article suggests a proven set of quality measures, but first, to define these measures, we must consider what a quality product looks like.

While software experts debate this point, every other field of engineering agrees on one basic characteristic of quality: A quality product contains few, if any, defects. In fact, the SEI has shown that this definition is equally true for software. We also know that software professionals who consistently produce defect-free or near defect-free products are proud of their work and that they strive to remove all the product's defects before they begin testing. Low defect content is one of the principal criteria the SEI uses for identifying the quality of software products.

Defining Process Quality

To define the needed quality measures, we must consider the fourth quality principle.

Quality Principle No. 4

The quality of a product is determined by the quality of the process used to develop it.

This implies that to manage product quality, we must manage the quality of the process used to develop that product. If a quality product has few if any defects, that means that a quality process must produce products having few if any defects. What kind of process would consistently produce products with few if any defects? Some argue that extensive testing is the only way to produce quality software, and others believe that extensive reviews and inspections are the answer. No single defect-removal method can be relied upon to produce high-quality software products. A high-quality process must use a broad spectrum of quality management methods. Examples are many kinds of testing, team inspections, personal design and code reviews, design analysis, defect tracking and analysis, and defect prevention.

One indicator of the quality of a process is the completeness of the defect management methods it employs. However, because the methods could be applied with varying effectiveness, a simple listing of the methods is not sufficient. So, given two processes that use similar defect-removal methods, how could you tell which one would produce the highest quality products? To determine this, you must determine how well these defect-removal methods were applied. That takes measurement and analysis.

The Filter View of Defect-Removal

This leads us to the next quality principle.

Quality Principle No. 5

Since a test removes only a fraction of a product's defects, to get a quality product out of test, you must put a quality product into test.

This principle also applies to every defect-removal method, from reviews and inspections, through all the tests and other quality verification methods. Every defect-removal method only removes a fraction of the defects in the product; so to understand the quality of a development process, you must understand the effectiveness of the defect-removal methods that were used. Further, to predict the quality of the delivered product, you must measure the effectiveness of every defect-removal step.

This also means that the highest quality development process would be the one that removed the highest percentage of the product's defects early in the process and then had the lowest number of defects in final testing. Finally, this means that the highest-quality products are those with the fewest defects on entry into the final stage of testing.

Criteria for a Quality Process

To evaluate a process, you must measure that process and then compare the measures with your criteria for a quality process. This means that you must have criteria that define what a quality process looks like. From the filter view of defect removal shown in Figure 1, we see that defect removal is like removing impurities from water [1]. To get water that is pure enough to drink, we should find progressively fewer impurities in each successive filtration step. Finally, if we were going to actually drink the water ourselves, we would not want to find any impurities in the final filtration step.

In effect, this means that the last filtration step is really used to verify the quality of the water produced by the prior stages. If there were any significant impurities, you would want to put that water through the entire filtration process again, starting from the very beginning. Then you might be willing to take a drink. Similarly, for a software system, this suggests three quality criteria.

1. Most of the defects must be found early in the development process.

2. Toward the end of the process, fewer defects should be found in each successive filtration stage.
3. The number of defects found in the final process stages must be fewer than some predefined minimum.

Determining Process Quality

While these sound like appropriate process-quality criteria, they have one major failing – you will not have complete defect data until the end of the process after the product has been built, tested, accepted, and used. During the process you will only know the number of defects found so far and not the number to be found in future stages. This is a problem because a low number of defects in a defect-removal stage could be because the product was of high quality, because the defect-removal stage was improperly performed, or because the defect data on that stage were incomplete. This means that you must have multiple ways to determine the effectiveness of a defect-removal stage and that these ways must include at least one way to evaluate the effectiveness of that stage at the time that it is actually enacted. Partial defect data can be used to do that. In fact, without these data, there is no way to determine the effectiveness of the defect-removal stages.

The three things we can measure about a process stage are: (1) the time the developers spent in that stage, (2) the number of defects removed in that stage, and (3) the size of the product produced by that stage. Then, using historical data, you could compare the data for any type of defect removal stage with like data for similar stages from previously completed projects. As long as you had comparable data for completed projects, you could see what an effective review, inspection, or test looks like. You could then determine the quality of each stage of the current project and either agree that the supplier proceed or repeat some prior phases until the quality criteria were met.

In-Process Quality Measures

From data on 3,240 Personal Software Process℠ (PSP℠) exercise programs written by experienced software developers, the SEI has determined the characteristics of a high-quality software process [1]. These data are shown in Table 1, and they show that developers inject about 2.0 defects per hour during detailed design and find about 3.3 defects per hour during detail-level-design reviews (DLDR).

To find the defects injected in one hour of design work, the average developer would have to spend 60*2/3.3 = 36 minutes reviewing that design. Similarly, since developers inject an average of about 4.6 defects per hour during coding and find about 6.0 defects per hour in code reviews, this same average developer should spend about 60*4.6/6 = 46 minutes reviewing the code produced in each hour. Since there is considerable variation among developers, the SEI has established the general guideline that developers personally spend at least half as much time reviewing design or code quality as they spent producing that design or code.

Further, from data on many programs, we have found that, when there are fewer than 10 defects found while compiling each 1,000 lines of code and fewer than 5.0 defects found while unit testing each 1,000 lines of code, that program is likely to have few if any remaining defects [2]. Combining these criteria with an additional requirement that developers spend at least as much time designing a program as they spent coding it, gives the following five software process quality criteria [1].

Calculating the Quality Profile

The quality profile has five terms that are derived from the data shown in Table 1. The equations for these terms are as follows.

1. Design/Code Time = Minimum(design time/coding time: 1.0).
2. Design Review Time = Minimum(2* design review time/design time: 1.0).
3. Code Review Time = Minimum(2* code review time/coding time: 1.0).
4. Compile Defects/KLOC = Minimum (20/(10 + compile defects/KLOC): 1.0).
5. Unit Test Defects/KLOC = Minimum (10/(5 + unit test defects/KLOC): 1.0).

To derive the five profile terms, consider formula No. 3 for code reviews. According to Table 1, in one hour of coding, a typical software developer will inject 4.6 defects. Since this developer can find and fix defects at the rate of 6.0 per hour, he or she needs to spend 4.6/6.0 = 0.7667 of an hour, or about 46 minutes, reviewing the code produced in one hour. Since there is wide variation in these injection and removal rates, and since the number 0.7667 is hard to remember, the SEI uses 0.5 as the factor. Based on experience to date, this has proven to be suitable. Since these parameter values are sensitive to application type and operational criticality, we suggest that organizations periodically analyze their own data and adjust these values accordingly.

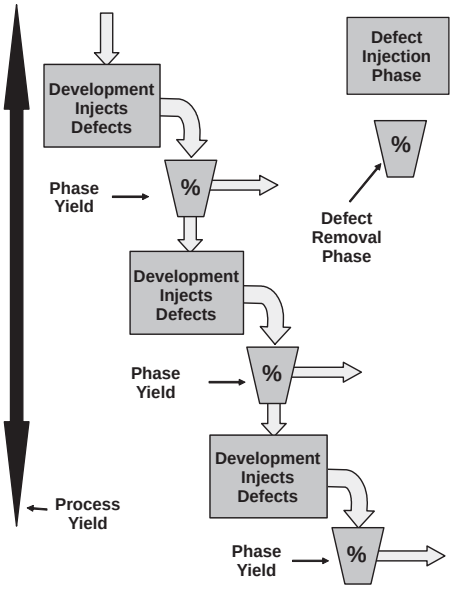


Figure 1: *The Defect-Removal Filtering Process*

The formula for the code review profile term compares the ratio of the actual time the developer spent reviewing code with the actual time spent in coding. If that ratio equals or is greater than 0.5, then the criteria are met. The factor of 2 in the equation is used to double both sides of this equation so it compares twice the ratio of review to coding time with 1.0. Also, to get a useful quality figure of merit, we need a measure that varies between 0 and 1.0, where 0 is very poor and 1.0 is good. Therefore, the equation's value should equal 1.0 whenever 2 times the code review time is equal to or greater than the coding time and be progressively less with lower reviewing times. This is the reason for the Minimum function in each equation, where Minimum(A:B) is the minimum of A and B. A little calculation will show that this is precisely the way equation No. 3 works. Equations No. 1 and No. 2 work in exactly the same way (except design time should equal or exceed coding time in equation No. 1).

To produce equations No. 4 and No. 5, the SEI used data it has gathered while training software developers for TSP

Table 1: *Defect Injection and Removal Rates (3,240 PSP Programs)*

Phase	Hours	Defects Injected	Defects Removed	Defects/Hour
Design	4,623.6	9,302		2.0
DLDR	1,452.7		4,824	3.3
Code	4,159.6	19,296		4.6
Code Review	1,780.4		10,758	6.0

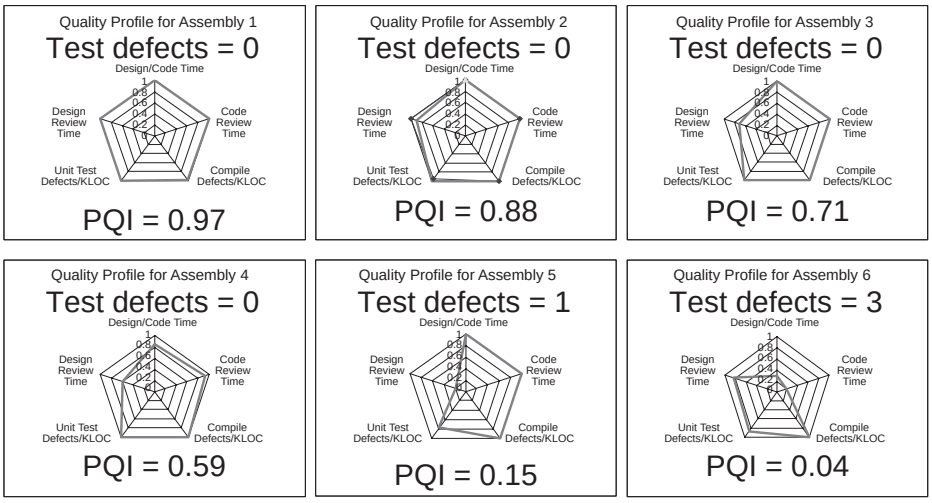


Figure 2: *Process Quality Profile (Six Programs)*

teams. It found that when more than about 10 defects/thousand lines of code (KLOC) were found in compiling, programs typically had poor code quality in testing, and when more than about five defects/KLOC were found in initial (or unit) testing, program quality was often poor in integration and system testing. Therefore, we seek an equation that will produce a value of 1.0 when fewer than 10 defects/KLOC are found in compiling, and we want this value to progressively decrease as more defects are found. A little calculation will show that this is precisely what equation No. 4 does. Equation No. 5 works the same way for the value of five defects/KLOC in unit testing.

One of the great advantages of these five criteria is that they can be determined at the time that process step is performed. Therefore, at the end of the design review for example, the developer can tell if he or she has met the design-review quality criteria. By plotting these five values on radar charts like those shown in Figure 2, it is relatively easy to identify a program's quality problems. The evaluation of these six profiles is as follows:

1. An excellent quality profile.
2. A similarly excellent quality profile.
3. A generally good quality profile with slightly too little design review time.
4. The design review measure is low, indicating potential problems that should be corrected with a repeated design review.
5. This product has a serious design review problem coupled with a unit testing problem. It should be reinspected. This product, when later tested had one defect found in final testing.
6. This product has serious design problems and an inadequate code review and should be replaced. This product had three defects found in subsequent testing.

Since these measures can all be available before integration and system test entry, and since they can be calculated for every component part of a large system, they provide the information needed to correct quality problems well before product delivery.

The Process Quality Index

For large products, it is customary to combine the data for all components into a composite system quality profile. Since the data for a few poor quality components could then be masked by the data for a large number of high quality components, it is important to have a way to identify any potentially defective system components. The process quality index (PQI) was devised for this purpose. It is calculated by multiplying together the five components of the quality profile to give a value between 0.0 and 1.0. Then the components with PQI values below some threshold can be quickly identified and reviewed to see which ones should be reinspected, reworked, or replaced.

Experience to date shows that, with PQI values above about 0.4, components typically have no defects found after development. Since the quality problems for large systems are normally caused by a relatively small number of defective components, the PQI measure permits acquisition groups to rapidly pinpoint the likely troublesome components and to require they be repaired or replaced prior to delivery. Once organizations have sufficient data, they should reexamine these criteria values and make appropriate adjustments.

Doing Quality Work

Since few software development groups currently gather the data required to use modern software quality management practices, we must consider the sixth principle of software quality.

Quality Principle No. 6

Quality products are only produced by motivated professionals who take pride in the quality of their work.

Because the measures required for quality management must be gathered by the software professionals themselves, these professionals must be motivated to gather and use the needed data. If they are not, they will either not gather the data or the data will not be very accurate. Experience shows that developers will only be motivated to gather and use data on their work if they use the data themselves, and if they believe that the practices required to consistently produce quality software products will help them do better work. Most developers who have used the TSP believe these things, but without proper training very few developers will.

While these measures and quality practices are not difficult, they represent a significant behavioral change for most practicing software professionals and their management. There are, however, a growing number of professionals who do practice these methods, and the SEI now has a program to transition these methods into general practice [1]. The methodology involved is the PSP, and to consistently use the PSP methods on a project, development groups must use the TSP. There is now considerable experience with these methods, and it shows that with proper use TSP teams typically produce defect-free or nearly defect-free products at or very close to their committed costs and schedules [2, 3, 4, 5].

Acquisition Pointers

Sound quality management is the key to software quality; without appropriate quality measures, it is impossible to manage the quality of a process or to predict the quality of the products that process produces. The developers must gather and analyze these data; they will not do this unless they know how to gather and how to use these data. This is why the sixth quality principle is critically important. Merely ordering the organization to provide the desired data will guarantee getting lots of numbers that are unlikely to be useful unless quality principle No. 6 is met. This requires motivating development management, and having development management train and motivate the developers in the needed quality measurement and management practices.

Once the developers regularly gather, analyze, and use these data, there only remains the question of how acquisition executives can get and use the data. This is both a contracting and a customer-supplier issue. Experience to date shows that when the developers use the TSP, you should have no trouble getting the required data [2, 3, 4, 5, 6, 7, 8].

The specific data needed to measure and manage software quality are the following:

1. The time spent in each phase of the development process. These times must be measured in minutes.
2. The number of defects found in each defect-removal phase of the process, including reviews, inspections, compiling, and testing.
3. The sizes of the products produced by each phase, typically in pages, database elements, or lines of code.

Planned and actual values are needed for these items, and these data should be for the smallest modules and components of the system. To establish and maintain the required management and developer motivation, these quality measurement and management requirements must be addressed both contractually and through management negotiation.

Conclusions

Poor quality performance damages a software development organization's cost and schedule performance and produces troublesome products. For acquirers to have a reasonable chance of changing the cost and schedule performance of their software vendors, they must demand effective quality management. The six principles of software quality reviewed in this article should help them do this.

By following these six principles and requiring suppliers to do so as well, you can consistently obtain quality software-intensive products at or very near to their committed costs and schedules.◆

References

1. Humphrey, Watts S. PSP: A Self-Improvement Process for Software Engineers. Reading, MA: Addison-Wesley, 2005.
2. Grojean, Carol A. "Microsoft's IT Organization Uses PSP/TSP to Achieve Engineering Excellence." CrossTalk Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Grojean.html>.
3. Davis, Noopur, and J. Mullaney. "Team Software Process (TSP) in Practice." Technical Report CMU/SEI-2003-TR-014. Pittsburgh, PA: Software Engineering Institute, Sept. 2003.
4. Humphrey, Watts S. Winning with Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.
5. Humphrey, Watts S. TSP: Leading a Development Team. Reading, MA: Addison-Wesley, 2006.
6. Rickets, Chris A. "A TSP Software Maintenance Life Cycle." CrossTalk Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Rickets.html>.
7. Trechter, Ray, and Iraj Hirmanpour. "Experiences With the TSP Technology Insertion." CrossTalk Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Trechter.html>.
8. Tuma, David, and David Webb. "Personal Earned Value: Why Projects Using the Team Software Process Consistently Meet Schedule Commitments." CrossTalk Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Tuma.html>.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI) of Carnegie Mellon University after retiring from IBM in 1986. He established the SEI's Process Program and led development of the Software Capability Maturity Model®, the Personal Software Process℠, and the Team Software Process℠. During his 27 years with IBM, he managed all of IBM's commercial software development and was vice president of Technical Development. He is an SEI Fellow, an Association for Computing Machinery member, an Institute of Electrical and Electronics Engineers Fellow, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. He has published several books and articles and holds five patents. In a White House ceremony, the president recently awarded him the National Medal of Technology. He has graduate degrees in physics and business administration.

Software Engineering Institute
4500 Fifth AVE
Pittsburgh, PA 15213-2612
Phone: (412) 268-6379
Fax: (412) 268-5758
E-mail: watts@sei.cmu.edu

Coming Events

January 4-7
Hawaii International Conference on System Sciences
Kauai, HI
www.hicss.hawaii.edu/HICSS39/apahome39.htm

January 8-10
IEEE Consumer Communications and Networking Conference
Las Vegas, NV
www.ieee-ccnc.org/ index.htm

January 8-11
Internet, Processing, Systems, and Interdisciplinaries (IPSI) USA 2006
Palo Alto, CA
www.internetconferences. net/california2006/index.html

January 11-13
The 33rd Annual Symposium on Principles of Programming Languages
Charleston, SC
www.cs.princeton.edu/~dpw/popl/06

February 6-9
Components for Military and Space Electronics Conference and Expo
Los Angeles, CA
www.cti-us.com/ucmsemain.htm

February 13-17
The Fifth International Conference on COTS-Based Software Systems
Orlando, FL
www.iccbss.org/2006

February 14-16
The International Association of Science and Technology for Development Conference on Software Engineering
Innsbruck, Austria
www.iasted.org/conferences/2006/Innsbruck/se.htm

May 1-4, 2006
2006 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

Role of Human Emotions in Requirements Management

Sreevalli Radhika. T.
Robert Bosch India Ltd.

Requirements management is an area where satisfactory results are not seen even after establishing well-defined processes and adopting good tools. This article looks at this problem from a totally different angle and finds facts that are normally not observed. It discusses the effect of human emotions in requirements management.

Many people stereotype software developers as emotionless individuals who have more in common with their computers than with their fellow workers. One thing brings out emotions in software developers more than anything else: requirements. Organizations try to establish defined processes for activities related to software development, and to achieve results that are fairly independent of various parameters (i.e., people, location, time) involved in accomplishing the desired activity. Generally, a reasonable success is assured from most of the processes with a few exceptions. One such exception is *requirements management*. This is because of the role emotions play in the acceptance of requirements. The effect of the emotional response to requirements can make a major impact on how software is developed.

Even after establishing a well-defined process and adopting a tool for managing requirements, it is often found that managing requirements is not enough. The main reason for this is understood with three human factors that generally affect the quality of any work:

- The way people act (work).
- The way people think.
- The way people feel.

The first two factors are normally noticed by project teams and are addressed by identifying required *tools* and *processes*. Tools and processes drive the actions and thinking process involved in the different phases of software development. They do not take into account the way people feel. Typically, there is no special attention given to this third factor. It is not apparent how many activities are directly affected by the emotional response of the developers. Most of the time, it is possible to achieve good results by taking care of the first two factors. However, even good results can be affected by emotions.

Other processes are not as affected by emotions; for example, consider configuration management. Configuration management usually does not provide opportunities where different people can *feel* differently about the handling of configuration items. There is not much room for different opinions to be formed in the way configuration items are identified and controlled. In general, configuration management requires specific actions rather than detailed consideration and analysis. As a result, it is not subject to the effects of emotions in the way that requirements management can be. Only the first factor mentioned above – the way in which people act – is important here and the differences in this factor can be avoided by defining a clear process for configuration management. Once a process is defined, the project team can just follow those steps. The *thinking* and *feeling* factors will come into play when considering how to improve the configuration management process.

The same is not true with requirements management. For this process, the thinking and feeling cannot be avoided. A well-defined process can only address so much, because there is something that affects the thoughts and feelings of the developers – *the requirements document*. The very existence of this particular document will have a continuous effect on the thoughts and feelings of the team members. The effect of this document continues through the life cycle regardless whether the document changes frequently, whether it is baselined, or whether it is maintained under a proper version control mechanism.

It is okay that the requirements document elicits emotions. A requirements document should have the power to enable the thoughts and feelings of the team members. What is important is the direction this document takes the reactions of the team members. It should help to bring innovation and motivation among the team members, rather than bringing irritation and frustration.

Studies about requirements documents show that the requirements usually have problems, including omissions, contradictions, ambiguities, duplications, inaccuracies, too much design, and irrelevant information. This may be true, but as far as the project team is concerned the emotions related to these issues are more serious than the fact that the problems exist. While processes may be in place to handle the requirements, they do not matter as much as the reactions of the developers.

It is easy to establish the physical traceability from requirements to all the affected documents in the project. Tools help in that task. However, the traceability from the author's intent to the final product does not occur as easily. Tools cannot help here. It is up to the relationship between the author and the reader. That relationship is a product of the respect the two have for each other and for the requirements document.

Lack of Respect for the Document

Almost everyone accepts the importance of communication and the effect it has on the human emotions and relations. If a day starts with a good incident, its effect remains the entire day. The same is true in case of the opposite situation. If the first incident of the day is bad, that effect lingers the rest of the day.

The same rule applies to the requirements document. When you consider the communication involved in a project, everything starts with the requirements document; the entire project depends on this document.

It is not only essential for this document to be clear and correct, but it should also gain the respect of the project team. If a requirement is clearly defined at this point, it will be accepted and respected by the software developers. If it is not, problems can ensue. Even something like a small contradiction between different sections of the document can deteriorate the respect for the document. This can be true even if it is a small contradiction that may not damage the clarity of the requirement as a whole.

If the requirements are not clear, the reader can start assuming things wherever a little bit of ambiguity exists. People tend to fill in the blanks. Often, they do not

even realize they are assuming things that are not there. In such situations, software developers will think they are capable of understanding a poorly written document. This reaction can cause problems down the road as assumptions can diverge from the original intent of the document.

A number of things can cause a lack of respect for the document. These include spelling mistakes, improper organization of the contents, and redundant statements. Authors may ignore small mistakes like the wrong date or the wrong version number. The readers will not. All of these items add up. If the readers disrespect the document, it can lead to frustration and anger. It will affect the actions they take in developing the requirements.

Some of these points may not seem important while preparing the document, but they can actually pass an indirect message to the reader about the document. The author should always consider the reaction of the reader. If the author takes care to avoid these little problems, it can bring about later benefits.

Lack of Respect for the Reader

A lack of respect for the reader ties closely to lack of respect for the document. If the author has a high respect for the reader, it will be apparent in every small part of the document. If the reader feels the respect of the author, he or she is more likely to accept the requirements and work with the author on future considerations.

An author can show his respect toward the reader in many ways. These include giving appropriate information at appropriate places and not leaving any loose ends. A reader who feels the document is giving him critical information – but not strict instructions – will feel more freedom in developing the requirements. This leaves the reader feeling he or she has an important contribution to make and is likely to gain project buy-in. Basically, the care the author takes in developing the document can make the reader feel better about the requirements document.

On the other hand, even a little carelessness in preparing the document can have a very negative effect. Things like improper formatting or inconsistency in font size can be as irritating as the errors mentioned above. If the reader feels the author was careless about these little things, he or she can feel a lack of respect from the author. This can create a reciprocal lack of respect for the document.

When the document is prepared with utmost care, it can demand respect from the reader. The reader will also feel respected. The reader will take more care in analyzing the document and will be more likely to work with the author to ensure everything is correct. That mutual respect can be a strong bond that will continue through the life of the project. Then the reader will more likely try to understand the document in detail and be more likely to cooperate with the author. Otherwise he or she will always see something wrong in the document.

When the reader does not have respect for the document or the way in which the document is written, then the reader's own point of view comes into play. It can often be very different from the requirements specified in the document. This can lead to a dislike or disregard for the author's point of view.

"Tools and processes drive the actions and thinking process involved in the different phases of software development. They do not take into account the way people feel."

Whoever the reader may be, once convinced of the quality of the requirement or lack thereof, the quality of the subsequent work will be affected by the reader's response to the requirement.

Lack of Respect for the Author

When the reader does not have enough confidence in the author who has prepared the specifications, the requirements will not get the consideration they need. Normally the author's background is not shown in the requirements document. However, the reader's past history with the author can color his or her reaction. A bad history increases the likelihood that he or she will expect difficulties and view the quality of the document with skepticism.

If readers have more technical knowledge than the author does, they may not read the document with the same point of view as the author. Such readers may quickly conclude that the document is not correct. Instead of finding the reason behind that, readers can assume that the requirement is wrong due to the author's ignorance and lack of skill.

Once the developers conclude that the author is not to be respected, the readers' analytical and technical capabilities will go in a direction of proving that the requirements are not feasible. Subsequent requirements from the same author are likely to be dismissed in the same way.

It is not that they intentionally consider the requirements this way, but it is difficult to overcome that bad initial reaction. Unless they see the reason, logic, and intention behind the requirement, they will never be able to succeed in implementing it as desired. If they are predisposed to disrespect the author, they are not likely to work with the author to ensure the requirements are implemented the right way.

All this does not mean that people should not give their comments on the document. The point here is that it should be done in a reasonable way. Putting some structure to the requirements review and analysis procedures will help with this, but will not solve all the problems caused by the lack of respect for the author. To understand the requirements, it is necessary to believe that there is some reason for them to be written as they are. To believe that such a reason exists, it is necessary to have respect for their creator. Cross training between the requirements' authors and the developers can help them understand each other's abilities and constraints. This can lead to more communication and understanding, which can only lead to better development and better results.

Two more things can bring out the emotions of the people involved: changes to the requirements, and the way in which change requests are handled.

Changes to the Requirements

Many times the freezing of requirements does not happen in time because of changes to the requirements after the project team has begun work. Scope creep happens, but it can lead to major frustration and even resentment on the part of the project team. If a lot of changes happen after the project team has started reading and reviewing the requirements, the team can lose faith in the project. This is especially true if changes to the requirements document take place after the design is started.

Most project teams see some risks and problems due to shifting requirements. If the requirements are not frozen and are changed many times, there may be lot of rework that, in turn, results in added effort and schedule variances. Unexpected

changes bring frustration and can lead to conflict between the requirements' author and the development team. It adds difficulty as the team tries to maintain various versions of the documents properly. Juggling the constant changes may lead to poor quality

However, these problems can be managed to some extent with the help of good processes and tools. But there are some aspects related to human emotions that should be considered more risky and difficult to handle.

As mentioned earlier, the project team should develop a liking or at least respectful acceptance toward the requirements. This will help the team achieve better design and a better quality work product. This is not possible unless the requirements are frozen. As long as there is a possibility to change the requirements document, there will be a feeling that the requirements can still be improved. It may also lead the project team to feel that the requirements will never be right.

Most developers have been on a project where there was a major delay in freezing the document. Let us use an example where development had to start when only 60 percent of the requirements were clear, and the requirements document was still undergoing changes. In such an instance, there can be a lot of suggestions for how to address the requirements. If the requirements were vague enough, they can bring out a number of suggestions. The team will then have to work through each of these alternatives to determine what was really desired by the requirements' author. Give and take with the author at this point can produce a series of requirement changes. It can be difficult to manage the suggestions raised within the project team when this occurs. As a result of their ability to make changes to the requirements, the developers will keep adding their desires to the requirements.

The problem here is that instead of spending the efforts on improving the designs, the project team spends its efforts on improving the requirements. No one assigned this task to them, but the team's frustration with the vague requirements will make them take it on themselves. This creates more work for them and for the requirements' author. The team spends more time on the requirements' developer's task than on their own assignments for no advantage. The rework adds time and cost to the project when it could have been avoided early on.

To alleviate this problem, the team should work with the requirements' developer to reach an understanding early on in the project. Requirements reviews can reduce the project team's frustration and help build a relationship between the author and the developers that can influence future projects. Once a solution is found, the requirements should be frozen and the changes limited to fixing problems.

The Way in Which Change Requests Are Handled

Regardless of the delay in freezing the requirements document, there will always be the possibility of getting change requests during the life cycle of the project. There are some difficulties that are specific to change request handling. These include capturing the change requests in a systematic way, evaluating the impact of these changes on the current development, and tracking them properly throughout the development life cycle. Mistakes in any of these tasks will be harmful to the project and are typically taken care of in the processes.

"A large number of changes bring out the frustration of developers because they feel they are trying to hit a moving target."

But again, the processes can only address the problems related to the first two human factors discussed in the beginning of this article. Mistakes in these kinds of tasks can happen if either the *action* or *thinking* of the team is not in a systematic or organized manner. These mistakes can also happen because of the feelings of the people involved. If the developers are already frustrated or irritated with the requirements document, their feelings can drive their behavior when handling changes to the requirements. If developers are found to be deviating from a process, it is necessary to consider not only the risks due to process considerations, but also the problems that may arise due to feelings.

If there is no process defined for handling the change requests or if the defined process is not followed consistently, then it will lead to various assumptions and negative feelings among the team members. People may take it as favoritism or injustice when the criteria for accepting or rejecting a change request are not apparent. If the behavior of the author or developers is not clearly understood, hard feelings may result. If the procedures and guidelines are not defined and practiced consistently, then the reason behind approving and rejecting the change requests will not be understood correctly by the team.

A large number of changes bring out the frustration of developers because they feel they are trying to hit a moving target. By the same token, if the changes are not handled properly, they can increase the level of frustration.

Conclusion

Human emotions play a very important role in requirements management. Developers react to requirements in a number of ways. The range of emotions they generate come from a variety of reasons. Poor requirements can elicit frustration, irritation, anger, and disrespect. Good requirements can bring acceptance, understanding, and buy-in. Processes can be defined for capturing, analyzing, reviewing, implementing, and verifying the requirements. Tools can be identified for tracking and implementing the requirements without errors. However, if human emotions and feelings are not addressed, then the desired result may not be achieved in spite of the use of those processes and tools.

Frustration and confusion over requirements can lead to unexpected behavior by development staff. When someone behaves in an unexpected way, people will question their behavior. There may be an attempt to change that behavior. But there should definitely be another consideration, which is *understanding the reason behind the current behavior*. Often, that reason is tied to the emotional response of the person.

How can an organization deal with emotional reactions to requirements problems? The best way is to take the emotions out of the process as much as possible. Many of these problems do not need a separate solution. Identification of the problem itself can lead to a quick solution in many cases. However, the following considerations can be adopted to improve the existing and established processes in this respect:

- A consistent and well-defined requirements definition process can help by taking the focus off the people and onto the process. This starts with a template for the requirements docu-

ment that covers information about the author and the reasons behind a particular requirement.

- Maximum care taken while preparing the requirements document will help avoid overlooking even the very small points like spelling mistakes that may deteriorate the respect on the document. Peer reviews can help catch these kinds of errors and ensure that the requirements document meets standards.
- The change request handling process must be clear, quick, and consistent. It should not be very easy to add a change request to the document. There should definitely be a three-step process like initiation, evaluation, and approval. These steps should be carried out quickly, but they should not be skipped.
- Involvement of the development team in reviewing the requirements early in the process will establish communications between the author and the developers. Open communication between the groups will make them all feel they are part of a team, and they are more likely to try to reach a mutually satisfactory result.

To achieve best results, the project team members should feel a part of the requirements process and should develop an involvement with the requirements. If they have a true stake in the results, their emotions will be guided toward achieving a common goal. Defined processes and support tools should complement their efforts and improve their possibilities for success.◆

About the Author

Sreevalli Radhika. T. works for Robert Bosch India Ltd. as a department quality coordinator. She has been involved in the management of embedded projects for the past decade. Radhika is also interested in fiction writing and has published more than 50 articles. She has a post-graduate degree in electronics and communication engineering.

Robert Bosch India Ltd.
Phone: (91-80) 2299-9052
Fax: (91-80) 2299-9156
E-mail: radhika.sreevalli@in.bosch.com
tsradhika@rediffmail.com

Letter to the Editor

Dear CrossTalk Editor,

Regarding the From the Sponsor article by Tom Christian in the August 2005 CrossTalk. The first paragraph of his article ends:

> "... relentless commitment to quality: employing peer reviews, configuration control, documentation, and testing."

Although these items are all necessary to achieve quality deliverable software, they are not sufficient: The one key item missing from the above list is, in my view, the most important one, good design practices.

I have heard said numerous times over the years, "You cannot test in quality," and it is so true. A team can spin its wheels for months thoroughly testing a system only to find itself retesting, retesting, retesting because every change seems to *break* the system in unintended ways. This is usually because the basic design of the system is flawed due to one or more of the following practices: use of global variables, lack of cohesion, close coupling, inadequate abstraction, lack of encapsulation techniques, etc. (All these principles I mention pre-date object orientation, yet it is surprising how little they are understood even today!)

A system with a truly good design could possibly succeed with limited testing, documentation, and peer reviews (configuration management is always crucial in my view). But, a poorly designed system will fail no matter how much it is tested, reviewed, or documented.

The larger and more complex the system, the more crucial it is to use sound design practices. It does not come automatically. No specific software language can guarantee it. It is a much larger challenge than "properly indenting your code." It is sorely needed today more than ever.

Robert Wolfman
Software Consultant, ITT Avionics

CROSSTALK
The Journal of Defense Software Engineering

ARTICLE INDEX VOLUME 18

TOPIC	ARTICLE TITLE	AUTHOR(S)	ISSUE	PAGE
Agile	Agile Software Development for the Entire Project	Granville Miller	12	9
	Extending Agile Methods: A Distributed Project and Organizational Improvement Perspective	Paul E. McMahon	5	18
Best Practices	Identifying Your Organization's Best Practices	David Herron, David Garmus	6	22
	The Myth of the Best Practices Silver Bullet	Michael W. Evans, Corinne Segura, Frank Doherty	9	14
Communication	Ports, Protocols, and Services Management Process for the Department of Defense	Dave R. Basel, Dana Foat, Cragin Shelton	5	6
Configuration Management	Configuration Management Fundamentals	Software Technology Support Center	7	10
	Finding CM in CMMI	Anne Mette Jonassen Hass	7	26
	Implementing Configuration Management for Software Testing Projects	Steve Boycan, Dr. Yuri Chernak	7	4
COTS	Security in a COTS-Based Software System	Arlene F. Minkiewicz	11	23
	Six Steps to a Successful COTS Implementation	Arlene F. Minkiewicz	8	17
Design	Dependency Models to Manage Software Architecture	Neeraj Sangal, Frank Waldman	11	8
	Effective Practices for Object-Oriented System Software Architecting	Ruth McCabe, Mike Polen	6	18
	How and Why to Use the Unified Modeling Language	Lynn Sanderfer	6	13
	Selecting Architecture Products for a Systems Development Program	Michael S. Russell	11	4
	UML 2.0-Based Systems Engineering Using a Model-Driven Development Approach	Dr. Hans-Peter Hoffman	11	17
	UML Design and Auto-Generated Code: Issues and Practical Solution	Ilya Lipkin, Dr. A. Kris Huber	11	13
Estimation	COCOMO Suite Methodology and Evolution	Dr. Barry Boehm, Ricardo Valerdi, Jo Ann Lane, A. Winsor Brown	4	20
	Creating Requirements-Based Estimates Before Requirements Are Complete	Carol A. Dekkers	4	13
	Estimating and Managing Project Scope for New Development	William Roetzheim	4	4
	Inside SEER-SEM	Lee Fischman, Karen McRitchie, Daniel D. Galorath	4	26
	A Method for Improving Developers' Software Size Estimates	Lawrence H. Putnam, Donald T. Putnam, Donald M. Beckett	4	16
	The Statistically Unreliable Nature of Lines of Code	Joe Schofield	4	29
	Software Cost Estimating Methods for Large Projects	Capers Jones	4	8
Information	Application-Specific Knowledge Bases	Dr. Babak Makkinejad	6	26
Interoperability	Software Component Interoperability	Jeffery Voas	11	28
Measurement	Balanced Scorecards From Golf to Business	Bill Ravensberg	8	27
	Measure Like a Fighter Pilot	Joe H. Lindley	9	19
	Performance-Based Earned Value	Paul J. Solomon	8	22
	Tying Project Measures to Performance Incentives	David P. Quinn	9	28
Miscellaneous	17th Annual Systems and Software Technology Conference Focused on Defense Capabilities		7	16
	Delivering Capabilities Through Partnerships	Chris D. Moore	8	8
	Handheld Computing	Col. Kenneth L. Alford, Ph.D.	6	4
	Technology Readiness Assessments for IT and IT-Enabled Systems	Robert Gold, David Jakubek	5	25
Open Source Software	Introduction to the User Interface Markup Langage	Jonathan E. Schuster	1	15
	Open Source Opens Opportunities for Army's Simulation System	Douglas J. Parsons, Dr. Robert L. Wittman Jr.	1	11
	Open Source Software: Opportunities and Challenges	David Tuma	1	6
	Opening Up Open Source	Michelle Levesque, Jason Montojo	1	26

CONTINUED ON NEXT PAGE

Monthly Columns:

ISSUE	COLUMN TITLE	AUTHOR
Issue 1: January Open Source Software	Publisher: Using Free Software Doesn't Mean It Won't Cost You Anything BackTalk: High Stakes and Misdemeanors	Elizabeth Starrett Gary A. Petersen
Issue 2: February Risk Management	Sponsor: Risk Management Offers Broad Payoffs Publisher: Exercising Risk Management Skills BackTalk: How Do You Make a Peanut Butter and Jelly Sandwich?	Kevin Stamey Tracy L. Stauder Mamie Danley Morgan
Issue 3: March Team Software Process	Sponsor: Team Software Process Brings Project Success Over Time Publisher: TSP Has Multiple Uses BackTalk: A Few Good Launch Coaches	Randy B. Hill Elizabeth Starrett Gary A. Petersen
Issue 4: April Cost Estimation	Sponsor: The Cost Estimation Conundrum Publisher: Increasing Confidence in Estimates BackTalk: How Much for the Elephants?	Thomas F. Christian Jr. Tracy L. Stauder Dr. David A. Cook
Issue 5: May Capabilities: Building, Protecting, Deploying	Publisher: Dr. Mom's Approach to Improved Capabilities BackTalk: Tackling Software Measurement? Try Proverbs.	Brent D. Baxter Carol A. Dekkers
Issue 6: June Reality Computing	Sponsor: Technology Fields Too Slowly Publisher: Computing Permeates Our Society BackTalk: You Want Reality Computing? You Can't Handle Reality Computing!	Kevin Stamey Elizabeth Starrett Dr. David A. Cook
Issue 7: July Configuration Management and Test	Sponsor: Processes Provide the Light of Successs Publisher: New Ways to Implement CM and Testing BackTalk: Software Plumbing	Randy B. Hill Tracy L. Stauder Tony Henderson
Issue 8: August Systems: Fielding Capabilities	Sponsor: Our Job Is to Get It There Publisher: Stay Focused on the User BackTalk: Bayonets and Deployment	Thomas F. Christian Jr. Elizabeth Starrett Dr. David A. Cook
Issue 9: September Top 5 Department of Defense Program Awards	Publisher: Integrated Teams and Sound Processes Bring Success BackTalk: A Model About Nothing	Tracy L. Stauder Gary A. Petersen
Issue 10: October Software Security	Sponsor: Software Security: Shifting the Pardigm From Patch Management to Software Assurance BackTalk: Network Passwords	Joe Jarzombek Dan Knauer
Issue 11: November Design	Sponsor: Software: Where We've Been and Where We're Going Publisher: Design Focuses on the How BackTalk: Design? We Don't Need No Stinkin' Design! (or "How to Fail Without Really Trying")	Kevin Stamey Tracy L. Stauder Dr. David A. Cook
Issue 12: December Total Creation of a Software Project	Sponsor: Successful Software Is an Epic Production Publisher: Software Development Is More Than Coding BackTalk: Push for Cheese: A Metaphor for Software Usability	Randy B. Hill Elizabeth Starrett Nicole Radziwill, Amy Shelton

Web Sites

Aeronautical Systems Center – Engineering Directorate Weapon System Software

https://www.en.wpafb.af.mil/software/software.asp

The Engineering Directorate of the Aeronautical Systems Center located at Wright-Patterson Air Force Base, Ohio, provides this informational site on weapon system software. The site provides updated guidance to address embedded software acquisition issues. This technical organization is charged with the duty and responsibility to provide all the engineering support necessary to maintain the world's finest Air Force, both today and into the future.

Systems and Software Consortium

www.software.org/ssci/default.asp

The Systems and Software Consortium (SSCI) was founded in the late 1980s to provide industry and government a resource for insight, advice, and tools that could help them address the complex and dynamic world of software and systems development. SSCI's focus is on delivering value by improving systems and software engineering tools and methods that members can apply to their programs to gain greater efficiencies and profitability. SSCI membership ranks include industry's Tier 1 market leaders as well as key government agencies and academic institutions.

Project Management Institute

www.pmi.org

The Project Management Institute (PMI) claims to be the world's leading not-for-profit project management professional association with more than 100,000 members in 125 countries. Members represent major industries, including aerospace, automotive, business management, construction, engineering, financial services, information technology, pharmaceuticals, healthcare, and telecommunications. PMI establishes project management standards and provides seminars, educational programs, and professional certification for project leaders.

Defense Contract Management Agency

www.dcma.mil

The Defense Contract Management Agency is the Department of Defense contract manager, responsible for ensuring federal acquisition programs, supplies, and services are delivered on time, at projected cost, and meet all performance requirements. The DCMA professionals serve as information brokers and in-plant representatives for military, federal, and allied government buying agencies – both during the initial stages of the acquisition cycle and throughout the life of the resulting contracts.

National Institute of Standards and Technology

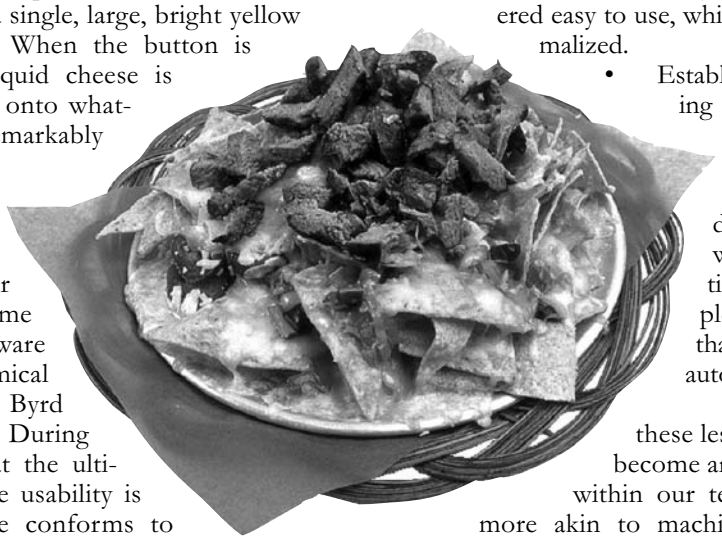
www.nist.org

The National Institute of Standards and Technology (NIST) is a non-regulatory federal agency within the U.S. Commerce Department's Technology Administration. NIST's mission is to develop and promote measurement, standards, and technology to enhance productivity, facilitate trade, and improve the quality of life.

Push for Cheese: A Metaphor for Software Usability

At the National Radio Astronomy Observatory's (NRAO) Science Center in Green Bank, W. Va., visitors curious about radio astronomy and the observatory's history and operations will discover an educational, entertaining experience. Employees also visit the science center, but their thoughts are more on afternoon snacks rather than distant galaxies. The employees of NRAO's Software Development Division in Green Bank have gained tremendous insight on the topic of software usability from many visits to the Science Center Café by pontificating upon the wisdom inherent in the design and use of the liquid cheese dispenser there.

The cheese dispenser is used to coat nacho chips in the familiar orange-tinted, viscous plasma prior to the addition of jalapeno peppers (or other toppers), intended to enhance the consumer's experience of the food product. The user interface is clear and unambiguous, with a single, large, bright yellow button labeled *Push for Cheese*. When the button is depressed, a stream of hot liquid cheese is expelled from the machine and onto whatever lies below. The cheese is remarkably consistent in its appearance, texture, and temperature. There is always only one variety of cheese dispensed.



Over the past three years, our team has spent considerable time and effort to improve the software systems used to make astronomical observations with the Robert C. Byrd Green Bank Telescope (GBT). During this time, we've discovered that the ultimate measure of GBT software usability is how well the user's experience conforms to what happens when they Push for Cheese. Our users want to press one button, have the software automatically interpret what they want the telescope to do, then see their results presented in a comprehensive, straightforward way. Though it's a lot to ask from software (especially since there are thousands of ways the GBT can be configured), similar concepts have been envisioned for years: In 1950, Turing argued that within five decades, computers would be intelligent [1]. In 1990, Newell presented his vision of a fully networked, intelligent environment in which machines and humans seamlessly interact (ubiquitous computing) [2]. At least in our environment, the knowledge embodied in the software is closely related to its perceived usability.

The International Organization for Standardization 9241 Part 11 defines usability as the "extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" [3]. Implicit in this definition is the notion of expectations because machines (and software) will only express the intelligence we seek. Who are our specified users? What is the context of use? How can we define and measure satisfaction? Here's what we've learned:

- Our users ultimately want a Push-for-Cheese user interface, where minimal and straightforward interactions with the software achieve the desired result.
- Conversely, users don't want the software to make too many assumptions on their behalf. As a result, more options may be required to achieve the balance between ease of use and software that's *too smart for its own good*. Push for Cheese assumes that the consumer wants liquid nacho cheese, which is a fine assumption for nachos, but not for a grilled cheese sandwich.
- Expectations are critical. Additional options can be designed into the system, but to add them, we must first establish requirements then develop, test, and deploy, which requires time and effort. Anyone who expects Brie or Swiss on their nachos when they Push for Cheese will be disappointed unless we've planned for other cheeses in advance.
- Even within a well-defined segment of specified users, there will still be substantial subjective variation in what's considered easy to use, which must be aggregated and normalized.
- Establish usability requirements during the analysis stages of a project to set bounds. A user expecting a graphical user interface won't like a command-line driven program; similarly, one who wants to specify many settings before execution will not be pleased with *intelligent* software that selects and tweaks settings automatically.

In addition to illustrating these lessons, Push for Cheese has also become an effective way to communicate within our team. When a usability feature more akin to machine intelligence or ubiquitous computing is requested, and is posited to us under the guise of ease of use, we console each other by affirming that they just want to Push for Cheese. In doing so, we compassionately acknowledge their desires, and then move to meet them halfway in implementation – while keeping the Push-for-Cheese vision for usability close to our hearts.

— Nicole Radziwill
— Amy Shelton
National Radio Astronomy Observatory
nradziwi@nrao.edu

References

1. Taatgen, N.A. "Poppering the Newell Test." Netherlands: University of Groningen, 2003 <www.ai.rug.nl/prepublications/BBSAlcom-NT-2003.pdf>.
2. Gray, J. 1999: "What Next? A Dozen Information Technology Research Goals." MSR-TR-99-50. Redmond, WA: Microsoft Research, June 1999 <http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-99-50>.
3. International Organization for Standardization. "ISO 9241 Part 11: Guidance on Usability." Geneva, Switzerland: ISO, 1998 <www.userfocus.co.uk/resources/iso9241/part11.html>.

CrossTalk is co-sponsored by the following organizations:

76th SOFTWARE MAINTENANCE GROUP

309th SOFTWARE MAINTENANCE GROUP

402d SOFTWARE MAINTENANCE GROUP

Homeland Security

CrossTalk / 309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737